8-2020

Leveraging Programmable Data Plane For Compressing Forwarding Tables

Garegin Grigoryan
gg5996@rit.edu

Leveraging Programmable Data Plane For Compressing Forwarding Tables

by

Garegin Grigoryan

A dissertation submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
in Computing and Information Sciences

B. Thomas Golisano College of Computing and
Information Sciences

Rochester Institute of Technology
Rochester, New York
August, 2020

Leveraging Programmable Data Plane For Compressing Forwarding Tables

by

Garegin Grigoryan

Committee Approval:

We, the undersigned committee members, certify that we have advised and/or supervised the candidate on the work described in this dissertation. We further certify that we have reviewed the dissertation manuscript and approve it in partial fulfillment of the requirements of the degree of Doctor of Philosophy in Computing and Information Sciences.

Dr. Minseok Kwon — Date
Dissertation Advisor

Dr. M. Mustafa Rafique — Date
Dissertation Committee Member

Dr. H.B. Acharya — Date
Dissertation Committee Member

Dr. Yaoqing Liu — Date
Dissertation Committee Member

Dr. Jayanti Venkataraman — Date
Dissertation Defense Chairperson

Certified by:

Dr. Pengcheng Shi — Date
Ph.D. Program Director, Computing and Information Sciences

Leveraging Programmable Data Plane For Compressing Forwarding Tables

by

Garegin Grigoryan

Submitted to the
B. Thomas Golisano College of Computing and Information Sciences Ph.D. Program in Computing and Information Sciences
in partial fulfillment of the requirements for the
Doctor of Philosophy Degree
at the Rochester Institute of Technology

Abstract

The Forwarding Information Base (FIB) resides in the data plane of a routing device and is used to forward packets to a next-hop, based on packets' destination IP addresses. The constant growth of a FIB forces network operators to spend more resources on maintaining memory with line-rate Longest Prefix Match (LPM) lookup in a FIB, namely, expensive and energy-hungry Ternary Content-Addressable Memory (TCAM) chips. In this work, we review two different approaches used to mitigate the FIB overflow problem. First, we investigate FIB aggregation, i.e., merging adjacent or overlapping routes with the same next-hop while preserving the forwarding behavior of a FIB. We propose a near-optimal algorithm, FIB Aggregation with Quick Selections (FAQS), that minimizes the FIB churn and speeds BGP update processing by more than twice. In the meantime, FAQS preserves a high compression ratio (at most 73%). FAQS handles BGP updates incrementally, without the need of re-aggregating the entire FIB table. Second, we investigate FIB (or route) caching, when TCAM holds only a portion of a FIB that carries most of the traffic. We leverage the emerging concept of the programmable data plane to propose a Programmable FIB Caching Architecture (PFCA), that allows cache-victim selection at the line rate and significantly reduces the FIB churn compared to FIB aggregation. PFCA achieves 99.8% cache-hit ratio with only 3.3% of the FIB placed in a FIB cache. Finally, we extend PFCA's design with a novel approach of integrating incremental FIB aggregation and FIB caching. Such integration needed to overcome cache hiding challenge when a less specific prefix in a cache hides a more specific prefix in a secondary FIB table, which leads to incorrect LPM matching at the cache. In Combined FIB Caching and Aggregation (CFCA), cache-hit ratio is maximized up to 99.94% with only 2.5% entries of the FIB, while the total number of route changes in TCAM is reduced by more than 40% compared to low-churn FIB aggregation techniques.

Acknowledgments

Research

I am very grateful to my first advisor, Dr. Yaoqing Liu for his constant help and motivation. Most of the research work presented in this document, such as FIB Aggregation with Quick Selections (FAQS) and the Programmable FIB caching Architecture (PFCA) was done under Dr. Liu's direct support and guidance. I hope to continue my collaboration with Dr. Liu as I consider it essential for my future research achievements.

I was very lucky to continue my Ph.D studies at Rochester Institute of Technology. My sincere gratitude goes to my advisor Dr. Minseok Kwon for supporting and guiding through new areas of research as well as helping me to restart my professional career. Specifically, I would like to thank Dr. Kwon for his insightful comments on writing a research paper and doing a research presentation.

General

It has been an exciting journey, that started at the Clarkson University of the North Country. My gratitude goes to the Computer Science faculty of Clarkson University, particularly, Dr. Christopher Lynch, Dr. Alexis Maciel, Dr. Christino Tamon, Dr. Jeanna Matthews.

At RIT, I had a great opportunity to learn from and work with wonderful faculty and staff - that includes Dr. Mohan Kumar, Dr. Pengcheng Shi, Dr. M. Mustafa Rafique, Dr. H. B. Acharya, Dr. Leon Reznik, Dr. Peizhao Hu, Lorrie Jo Turner, Min-Hong Fu, Amanda Zeluff.

I am extremely grateful to my parents for supporting me from the very first day.

Thank you Alan Beadle for the support and motivation.

Thank you Ashok Babel for being a great and reliable friend.

Thanks to John Marshall and James Coole for supporting me during my Cisco internship. Special thanks go to my friends and colleagues: Alain Lafaye, Behman Mofakham, Suman Bhowmick, Soham Dongargaonkar, Igor Khokhlov, Sahil Gupta, Nibesh Shrestha, Avinash Maurya, Andrei Aleksandrov, Eleanor Kellam, Matthiew Turner, Patrick Jaouen, Jeffrey D. Carr, Susan Gemmett, Mathieu Wong, Keivan Bahmani, Steve Liebich, Paul D. Holt, Austin Jantzi, Rasool Mazruee, Masoud Moghaddam, Hamid Eisazadeh, William van Doorn.

To my parents, Hasmik and Alexandr Grigoryan, and my brother, Artashes Grigoryan.

Preface

This thesis is based on the following conference papers and journal articles:

1. Grigoryan, G., Liu, Y., Kwon, M. 2020. PFCA: a programmable FIB caching architecture. To appear in IEEE/ACM Transactions on Networking (ToN).
2. Grigoryan, G., Liu, Y., Kwon, M., 2020. Boosting FIB Caching Performance with Aggregation. In Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing. 2020, ACM [31].
3. Liu, Y., Grigoryan, G., 2018, November. Toward incremental FIB aggregation with quick selections (FAQS). In Proceedings of 17th International Symposium on Network Computing and Applications (NCA) (pp. 1-8). 2018, IEEE [46];
4. Grigoryan, G., Liu, Y., 2018. PFCA: a programmable FIB caching architecture. In Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems (pp. 97-103). 2018, ACM [30].
5. Grigoryan G., Liu Y. "Toward a programmable FIB caching architecture," In Proceedings of IEEE 25th International Conference on Network Protocols (ICNP), Toronto, ON, 2017, pp. 1-2 [29].

The co-authors of the listed papers gave their permission to use the papers for this document.

Abbreviations

AS	Autonomous System
BT	Binary Tree
CFCA	Combined FIB Caching and Aggregation
DRAM	Dynamic Random Access Memory
FAQS	FIB Aggregation with Quick Selections
FIB	Forwarding Information Base
ISP	Internet Service Provider
LPM	Longest Prefix Match
LTHD	Light Traffic Hitters Detection module
PFCA	A Programmable FIB Caching Architecture
PSA	Portable Switch Architecture
PT	Patricia Tree
RIB	Routing Information Base
RM	Route Manager
SRAM	Static Random Access Memory
TCAM	Ternary Content-Addressable Memory

Contents

List of Figures

List of Tables

Chapter 1

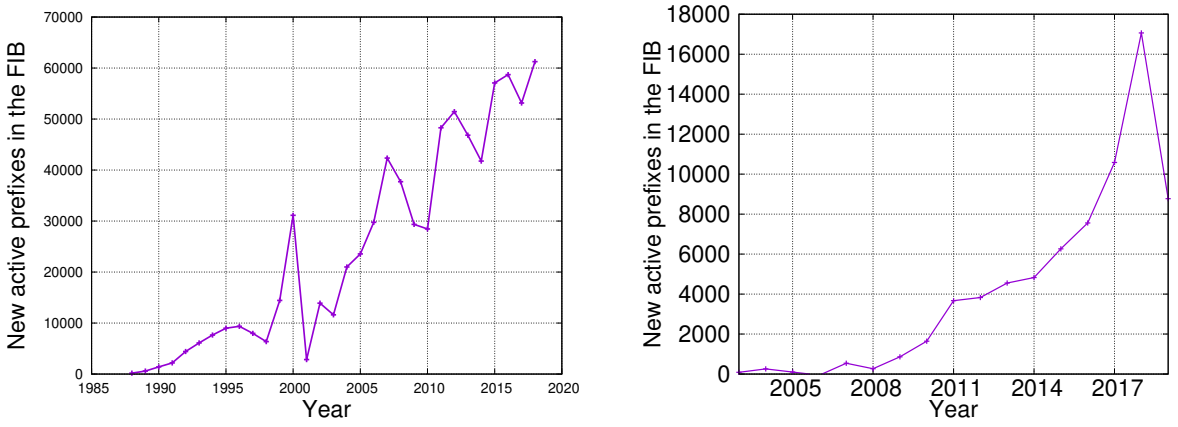
Introduction

1.1 Problem overview

A Forwarding Information Base (FIB) is used for IP prefix lookup and packet forwarding based on a packet's destination IP address. FIBs in backbone routers of many Internet Service Providers (ISPs) contain more than 800,000 IPv4 and 70,000 IPv6 entries and the number of entries continues to increase [8] (see Figure 1.1). To achieve fast prefix lookups while overcoming Longest Prefix Match (LPM) rule challenges, FIBs are usually installed in Ternary Content-Addressable Memory (TCAM) chips of modern high-end routers. Unlike Dynamic Random-Access Memory (DRAM) or Static Random-Access Memory (SRAM) with storage as a primary function, TCAM is a unique type of hardware solution that provides one-CPU-cycle parallel search over hundreds of thousands of FIB entries [28]. However, TCAM chips are significantly more expensive and energy-consuming than SRAM or DRAM chips [16, 37, 44, 54]. Maintaining power-hungry TCAM chips significantly increases operators' production costs for TCAM usage, including such expenses as providing cooling facilities for the networking hardware [6, 71]. Moreover, the continuous growth of the global FIB size and widespread adoption of IPv6 may lead to TCAM overflow problem and thereby the Internet service disruptions [3]. These potential consequences drive network operators to buy new high-capacity TCAM chips with higher cost and additional energy consumption or limit a memory allocation for IPv6 routes [20]. Based on our market analysis, the price of the Cisco ASR 9000 Series Line Card that can support up to 1M of TCAM entries can make up to 80% of the total cost of a router.

The super-linear growth of global FIB can be attributed to two main factors:

1. *The growth of Internet users across the world.* Since the beginning of 2018, the Internet Assigned



(a) IPv4 prefixes

(b) IPv6 prefixes

Figure 1.1: BGP table growth.

Numbers Authority (IANA) allocated more than four thousands of new Autonomous System Numbers (ASNs). The widespread deployment of IPv6 will trigger ASN and prefix allocations even more.

2. *Traffic-engineering and multi-homing used to provide fine-grained routing* [80]. Both techniques imply prefix fragmentation or advertising more specific prefixes via BGP (Border Gateway Protocol). Studies show that 50% of prefixes announced by BGP are more specific prefixes [53].

There are several approaches targeting TCAM overflow and FIB growth problems. Zhao *et al.* in [80] put them into two broad categories, namely, long-term and short-term solutions. The *long-term solutions* [51,66] include revision of the business relations between Autonomous Systems, e.g., network operators working in the Default Free Zone (DFZ) can be compensated for keeping all routes in a FIB. Alternatively, re-design of the routing architecture, e.g., splitting address space into a locator (for routing systems) and an identifier (for end systems), may greatly reduce the size of the global FIB table. However, both strategies require a coordinated deployment on a global scale. In this thesis, we investigate *short-term solutions*, that can be applied locally in an Autonomous System (AS) and significantly prolong the lifetime of a TCAM chip. First, we study FIB aggregation, which network operators believe to be one of the most feasible solutions as it has a clear deployment strategy and benefit [80]. Next, we study FIB caching, that greatly compresses the FIB on TCAM by offloading it from unpopular routes.

1.2 FIB aggregation

FIB aggregation [23, 46, 48, 72] merges prefixes that share the same next-hop in a FIB. The resulting compressed FIB should maintain the same forwarding behavior as the original FIB. That is, a packet with an arbitrary destination IP address should be forwarded to the same next-hop (or dropped) by a compressed FIB as with the original FIB. In the meantime, a FIB aggregation algorithm must correctly and efficiently handle numerous route updates, advertised through Border Gateway Protocol (BGP). FIB aggregation algorithms, based on Optimal Routing Table Constructor (ORTC) [23, 48], may compress a FIB by at most 80% depending on the number of neighbors an AS has. However, those algorithms suffer from several drawbacks:

1. *High time costs for processing route updates, including additions, withdrawals, and changes (i.e., BGP updates).* For instance, one of the state-of-the-art FIB aggregation algorithms, FIFA-S [48] achieves optimal aggregation ratio for each update but needs to perform two subtree traversals in the control plane for such a goal.

2. *High BGP churn.* Individual BGP updates result in a burst of changes in a FIB. An increase of writes into a FIB is especially disadvantageous for TCAM chips, where an entry insertion may require up to 1000 operations [34].

3. *Memory consumption at the control plane.* The optimal compression ratio in ORTC is achieved at the expense of high memory usage [76]: each node generated by the aggregation algorithm in the control plane contains an array of a variable size, which stores next-hop candidates to be used for next-hop selection for aggregated prefixes.

1.3 FIB caching

An alternative to FIB aggregation, FIB caching utilizes the high skewness between the number of popular and unpopular routes in a global FIB. In fact, merely 1.93% of FIB entries cover more than 99.5% of flows going through ISP network [25]. FIB caching addresses this important property of the Internet traffic by storing the most popular routes in the expensive memory (such as TCAM) and relocating unpopular routes to cheaper memory units (such as DRAM). The switch hardware manufacturers widely applied FIB caching until the early 1990s. Specifically, Cisco's *fast switching* [18] implied storing the recently used prefixes in faster memory. While FIB caching has several advantages over FIB aggregation, such as a smaller FIB size and a smaller BGP churn on TCAM, several flaws of current FIB caching architectures impede its

widespread deployment. First, increased amounts of a cache miss cause packet losses at slow memory buffers of the data plane [41]. Second, FIB caching solutions lack an efficient procedure to select victim cache entries for cache replacement. Random cache victim selection may result in evictions of popular prefixes from the cache. Thus, popularity-based victim selection is more reliable and accurate. It was shown that LRU (Least Recently Used) policy is the most efficient replacement algorithm [25, 41]. However, LRU is a software solution and is impractical for implementing on high-end routers with large routing tables [47] due to the performance concern. In [47], Liu *et al.* proposed to use idle timeouts to evict the least popular entries. Yet, such a strategy may lead to TCAM overflow when the number of new cache entries exceeds the number of idle cached flows.

1.4 Our contribution

First, we propose a near-optimal algorithm, FIB Aggregation with Quick Selections (FAQS), that doubles the speed of BGP update processing and minimizes BGP churn in the data plane. Different from existing aggregation algorithms, FAQS uses a single tree traversal to conduct both FIB aggregation and handle FIB updates. It handles routing updates incrementally, without re-aggregation of an entire forwarding table. On a single BGP update, in the worst case, FAQS will traverse only a subtree rooted at the updated node and its parents' nodes. Furthermore, unlike FIFA-S, FAQS keeps only a single next-hop at each node and considerably reduces memory consumption for aggregation operations. The outcome of our improvements is the significant acceleration of FIB aggregation and update handling. Although FAQS is still a heuristic aggregation algorithm, we experimentally prove its superior performance via multiple realistic datasets in different Routing Information Bases (RIBs) with more than 1 billion route updates from Route Views Project [5] over a 6-year period. The results are briefly described as follows:

1. FAQS achieves high and near-optimal compression ratios: ***reducing the number of FIB entries by up to 73% for IPv4 and 42% for IPv6***.
2. FAQS runs up to ***2.53 and 1.75 times as fast*** as the existing ORTC algorithms for IPv4 and IPv6 FIBs, respectively.
3. FAQS reduces the ***average number of FIB changes by 30% for IPv4 routing tables and by 10% for IPv6 routing tables***.
4. FAQS can ***save up to 30% of memory consumption*** compared with the algorithms that achieve optimal aggregation ratio.

Second, we revisit the idea of FIB caching (or route caching) from a different perspective from traditional approaches. We propose to leverage the programmable data plane, namely, Protocol Independent Switch Architecture (PISA) and P4 language [2,13] to minimize cache-miss latency and enhance cache replacement performance. We call our new design a Programmable FIB Caching Architecture (PFCA). In PFCA, we adopt several features of PISA and P4. First, we take advantage of the programmability of match-action tables to design two levels of caching structure. Second, we employ the programmability of the ingress pipeline in PISA, which allows PFCA to process cache-miss packets at the line-rate. Third, PFCA uses P4 registers, counters and hash functions to identify unpopular and popular routes, trigger cache replacement operations and conduct pipeline-based packet processing.

In summary, our key contributions in PFCA are as follows:

1. We designed PFCA, a programmable FIB caching architecture, where cache-miss operations are handled entirely in the data plane.
2. We designed a pipeline-based mechanism for cache victim selection and emulated it in a virtual P4 switch.
3. Our evaluation shows that PFCA uses ***3.34% entries (20K) of the full FIB to achieve 99.825% hit ratio for Level-1 cache, and only 0.014% of the total traffic is redirected to the slow memory*** thanks to the Level-2 cache with 6.68% (40K) entries of the full FIB. In addition, we show a significant deduction of expensive writing operations on TCAM caused by BGP routes updates.

Third, we **integrated FIB aggregation and caching** for small FIB size with efficient use of TCAM and a higher cache-hit ratio, but not compromising LPM matching correctness. To this end, we design *Combined FIB Caching and Aggregation* (CFCA in short) by introducing an aggregation layer into the *Route Manager* of the control plane in PFCA. Unlike other FIB aggregation algorithms [23, 46], CFCA does not produce overlapping routes, and thus prevents *cache hiding*[1], while handling BGP updates incrementally without re-aggregation of the entire FIB. Specifically, CFCA uses prefix binary trees and prevents prefixes from the same branch to be installed into the cache and the main FIB. CFCA's aggregation frees up significant space, dramatically increases cache-hit ratios, and stabilizes the cache compared to other FIB compression mechanisms. Our results show that when the size of a FIB on TCAM is only 2.50% of a full FIB, TCAM miss ratio is reduced by 80% while the churn is reduced by 40% in comparison to standard FIB aggregation, indicating that both FIB aggregation and caching in CFCA mutually boost their advantages but eliminate the disadvantages when applied together.

[1] Hiding of a longer matching prefix in the main memory, while a shorter matching prefix stays in a cache (See Section 4.1)

Our contributions with CFCA are as follows:

1. We design a FIB aggregation algorithm compatible with FIB caching. CFCA's aggregation preserves the forwarding behavior of a FIB without introducing overlapping routes. As a result, CFCA outperforms existing FIB caching techniques by achieving ***99.94% cache-hit ratio with only 2.50% of the FIB entries in a TCAM cache***.

2. We design CFCA's incremental BGP update handling algorithm. In CFCA, only ***0.625% of BGP updates trigger changes in TCAM***; the *total churn in TCAM is reduced by 40%* compared to a standard FIB aggregation. In the meantime, the time cost for BGP update handling by CFCA exceeds PFCA's by only 0.09μs (11.6%).

3. We evaluate CFCA using traffic traces with ***more than 3.5 billion of packets and 45,600 BGP updates***. Our simulation demonstrates the advantages of CFCA over standard FIB caching and aggregation approaches in terms of cache-miss ratio and cache churn.

1.5 Organization

In Chapter 2, we give the necessary background on routing, LPM matching, and FIB compression. In Chapter 3, we study FIB aggregation and propose FAQS, a faster FIB aggregation algorithm with the minimized churn. In Chapter 4, we study FIB caching and propose PFCA, a Programmable Data Plane Architecture for efficient FIB caching. In Chapter 5, we present CFCA, Combined FIB Caching and Aggregation for boosting FIB compression while reducing cache-miss rations and data plane churn. In Chapter 6, we review the related work in the area of FIB aggregation, caching and forwarding equivalence verification. Finally, in Chapter 7, we conclude this thesis and discuss future work.

Chapter 2

Background

2.1 Routing basics

Routers are the key devices for computer networking. The **control plane** of a router collects and exchanges paths towards other network destinations. The **data plane** of a router stores the routing table and forwards packets accordingly to it.

The global Internet routing space is divided into Autonomous Systems (ASes) - sets of routers under a single administration [33]. The ASes are allocated with blocks of IP addresses, unified into IP prefixes. While intra-AS routing is regulated by Interior Gateway Protocols, defined internally by AS operators, the inter-routing between ASes is conducted through the external Border Gateway Protocol (BGP). BGP protocol allows an AS to advertise its network (i.e., its IP prefixes) and share paths to other networks with neighbor ASes. In addition, each AS designs its own BGP policies for path selection and advertising. BGP peer sessions result in a Routing Information Base (RIB), stored at the control plane of a router. A RIB contains the known prefixes and paths towards them. In standard routers, these prefixes and the first hops from the corresponding paths are directly installed into a Forwarding Information Base (FIB) of a router's data plane. When a packet arrives at a router, its destination IP address should be matched against the FIB in order to obtain the next-hop, where the packet will be forwarded by the router's forwarding engine. The IP matching at the data plane adopts the Longest Prefix Matching (LPM) rule, discussed in the following section.

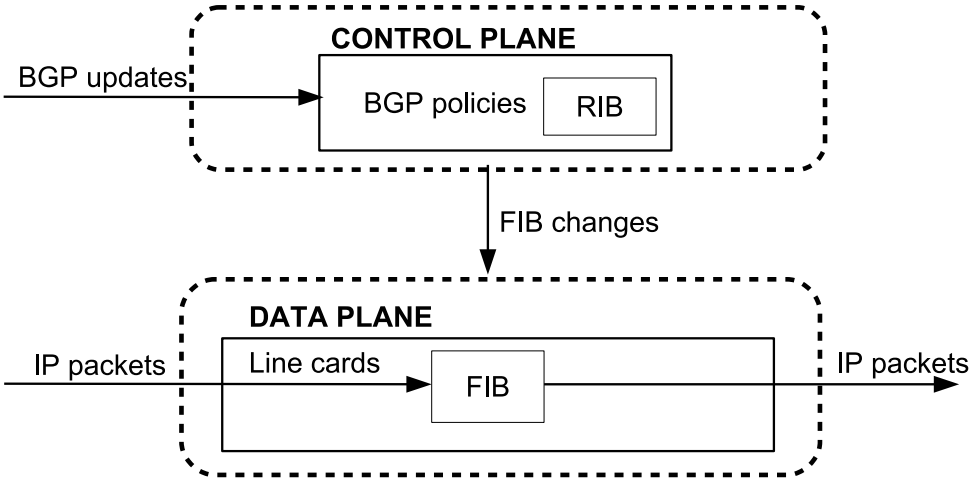


Figure 2.1: The architecture of a traditional router running BGP protocol.

2.2 Longest Prefix Match (LPM) rule

According to the Longest Prefix Match rule, when an IP packet is matched against an FIB, among the prefixes matching its destination IP address the longest one must be selected. We give the formal definition below.

Definition 1. Suppose p is a prefix with length l_p in a forwarding table T. We denote p as $p_1p_2..p_{l_p}$, where p=$\{0,1\}^{l_p}$ (i.e., p_i is 0 or 1 for $i = 0, 1, 2, ..., l_p$). Also suppose there is a string $s = \{0,1\}^{l_s}$, where l_s is the length of s. Then, according to the Longest Prefix Matching rule, we define that p is the ***Longest Prefix Match*** for s in T, namely, $p = LPM_s(T)$, if and only if

1. $l_p \leq l_s$
2. p is a prefix for s, i.e. $p_1p_2..p_{l_p} = s_1s_2..s_{l_p}$.
3. $\nexists\, p'$ in T, where p' is a prefix of s and $l_{p'} > l_p$.

We use Table 2.1 to illustrate the LPM selection. The table represents a sample FIB for IPv4 addresses with 32-bit address space. Several cases may happen during the matching process:

1. An IP destination address does not have a match in an FIB. In such a case, the packet will be dropped by the router. For example, an IP destination address that does not start with the prefix *128.153.0.0/16*, for example, *45.56.76.120*, will be discarded.

Prefix	Next Hop
128.153.0.0/16	1
128.153.64.0/18	2
128.153.128.0/17	3
128.153.192.0/18	4
128.153.96.0/19	5

Table 2.1: Forwarding Information Base (FIB).

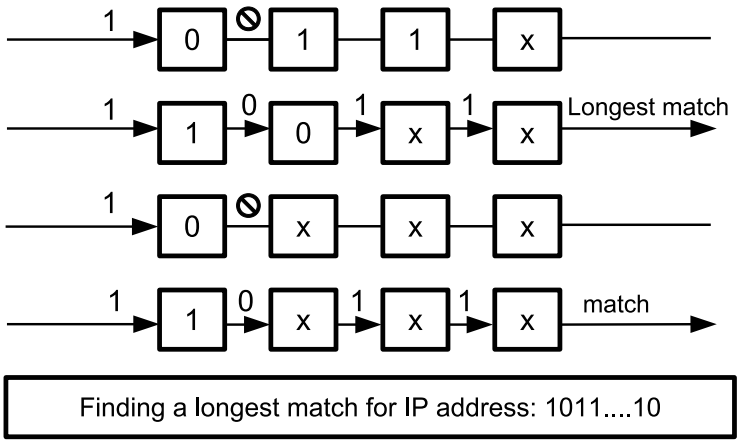


Figure 2.2: Longest Prefix Matching in TCAM.

2. An IP destination address has a single match in a FIB. In such a case, the packet will be simply forwarded to the corresponding next hop. An example of such an IP address is *128.153.0.11* with the match *128.153.0.0/16* and next-hop 1.

3. An IP destination address has several matches in the table. In such a case, a match with the longest prefix length will be selected. An example of such an IP address is *128.153.124.35*, that matches prefixes *128.153.0.0/16*, *128.153.64.0/18*, *128.153.96.0/19*. However, only the prefix *128.153.96.0/19* will be selected by the data plane engine, since it is the Longest Prefix Match. Thus, the packet will be forwarded to the next-hop 5.

2.3 LPM at the data plane

The amount of traffic at backbone networks may reach up to 170 Tbps in 2021 [24]. Such amounts of traffic require backbone routers to perform the LPM at a line rate. This goal is achieved by placing a FIB

in Ternary-Content Addressable Memory (TCAM) chips. Entries in TCAM should be sorted by their prefix lengths, starting with the longest prefix. The bits of an IP address are matched against all the prefixes in parallel, and the highest in position matching prefix is selected.

The design of TCAM makes it significantly more expensive than Static Random Access Memory (SRAM) or Dynamic Random Access Memory (DRAM) chips [16, 37, 54]. The cost of storage per single bit in TCAM is about 30 times more expensive than in SRAM due to the greater number of transistors [16], [54]. While a TCAM chip occupies much larger physical space than SRAM [52], the number of entries that TCAM is able to maintain is smaller compared to both SRAM and DRAM. The size of TCAM available on the market in 2016 was ranging from 1Mb to 72Mb [45]. Due to the high power demands of TCAM chips, the smaller capacity TCAM are the most popular on the market. Meanwhile, the constant increase of prefixes in the global FIB leads to TCAM overflow problem and forces network operators to upgrade their forwarding devices with bigger-size TCAM [80].

In this work, we leverage different FIB compression techniques to mitigate TCAM overflow problem. In the following chapter, we present FAQS, a FIB aggregation with Quick Selection, that compresses the forwarding table at the smaller cost than previous FIB aggregation techniques.

Chapter 3

FIB Aggregation with Quick Selections

3.1 Introduction

FIB aggregation refers to a process, that merges FIB entries with different prefixes and the same next-hop. The rationale is that as long as data packets can be forwarded with the correct next-hop and reach their destinations correctly, it is irrelevant using which route, either the original one or an aggregated one with a different prefix length. While FIB aggregation may greatly compress the size of a FIB, the aggregation process must ensure 100% forwarding correctness and should not change the forwarding behaviors for any packet. In other words, the original and aggregated FIB tables should be *Forwarding Equivalent*.

In the following definition, we denote:

1. $LPM_\omega(T)$ as a Longest Prefix Match of an IP address ω in a table T (see Definition 1 in Section 2.2 for a formal definition of a Longest Prefix Match).
2. $N_T(p)$ as a next-hop for a prefix p in a table T.

Definition 2. The forwarding tables $T_1, T_2, ..., T_m$ are *Forwarding Equivalent*, if and only if, for every single IP address $\omega = \{0,1\}^n$, $N_{T_1}(LPM_\omega(T_1)) = N_{T_2}(LPM_\omega(T_2)) = ... = N_{T_m}(LPM_\omega(T_m))$, where n is the length of an IP address.

In Table 3.1a FIB entries B and C have the same next-hop value as the entry A, which fully covers IP address blocks of both B and C. Hence, excluding entries B and C from FIB table will not change the forwarding behaviors for any packets matching against B or C, which preserves the *Forwarding Equivalence* of the

Table 3.1: FIB aggregation process.

(a) Original FIB table

Label	Prefix	Next-hop
A	141.92.0.0/16	1
B	141.92.64.0/18	1
C	141.92.0.0/19	1
D	141.92.192.0/19	2
E	141.92.224.0/19	2

(b) Compressed FIB table

Label	Prefix	Next-hop
A	141.92.0.0/16	1
D	141.92.192.0/19	2
E	141.92.224.0/19	2

original and aggregated tables. Excluding entries D or E, in contrast, will not preserve the *Forwarding Equivalence*, e.g., packets with destination IP addresses from these blocks will be forwarded to the next-hop 1 instead of 2.

A compressed FIB after aggregation is given in Table 3.1b with 3 entries, which yields the same forwarding behaviors as the original unaggregated FIB (Table 3.1a). While this is a simple example of correct FIB aggregation, more complicated cases need to be handled by an efficient aggregation algorithm. These cases include handling overlapping routes, where the majority of prefixes share the same next-hops, however, certain prefixes have different next-hops. Finally, FIB aggregation complicates BGP update handling, since these updates are designated for original prefixes, rather than aggregated prefixes.

FIB aggregation module needs to operate at the control plane of a routing device, to compress the FIB before pushing its entries into the data plane. Next, we describe the data structures we utilize for our proposed *FAQS* algorithm.

3.2 Data structures

To aggregate real FIB tables with hundreds of thousand entries, the control plane's aggregation algorithm needs to utilize compact and efficient data structures. In this work, we leverage PATRICIA tree (PT) [55] to store FIB entries. Similarly to a prefix binary tree, the nodes in such a tree correspond to prefixes (or bit strings). More specifically, the root node corresponds to the prefix with the length 0; the left or the right

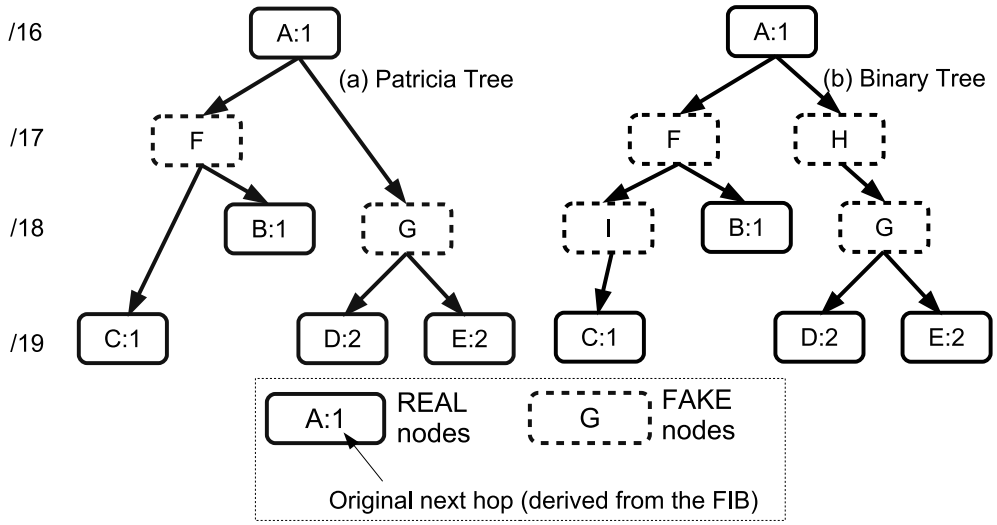


Figure 3.1: PT and BT for FIB entries from Table 3.1a.

child of a node corresponds to a prefix with an appended bit 0 or 1 respectively. However, unlike a full Binary Tree (BT), which requires the prefix length difference between a parent and child node to be exactly one, the difference in a PT can be more than one. We illustrate the differences between a PT and a BT representation in Figure 3.1 for the FIB in Table 3.1. Note that we call the nodes derived from the original FIB table as *REAL* nodes, while ancillary nodes, required to bind real nodes are called as *FAKE* nodes.

In a BT, looking up the next-hop for a given destination address is based on its bits: starting from the first bit, if a bit is 0, then move to the left branch of the BT, otherwise, move to the right. This is a recursive process until an LPM prefix is found. In a PT, movements also stem from the difference between nodes' levels and the bit values that reflect the branch that connects two nodes, however, a single movement can include several bits instead of a single bit.

As we can observe, the use of a PT significantly reduces memory consumption and the number of memory access times for a prefix search. This feature is especially critical for compact representation of IPv6 routing tables which have 128-bit address space.

3.3 Design

3.3.1 Static FIB aggregation

As it was mentioned above, our proposed algorithm, FIB Aggregation with Quick Selections (*FAQS*) uses a data structure based on the PATRICIA tree (PT) [55]. Each node in a PT has the following fields (we denote a node as n):

1. *Node type*, denoted by *T(n)*. If a node was derived from an original FIB entry, the value is *REAL*; otherwise, if a PT node is only an ancillary node that helps to form the PT, the value is *FAKE*. In Figure 3.2(a), *T(F)* and *T(G)* are *FAKE*, and all the other nodes' types, i.e., *T(A)*, *T(B)*, *T(C)*, *T(D)*, *T(E)* are *REAL*.

2. *Original next-hop*. The next-hop value that is associated with an original FIB prefix and mapped to a PT node, denoted by *O(n)*. For a *REAL* node, it is taken from the FIB; for a *FAKE* node, it is derived from the original hop of its nearest *REAL* ancestor node during the top-down instantiation described below. This instantiation process is carried out during aggregation.

3. *Selected next-hop*. The next-hop value of a prefix after aggregation, denoted by *S(n)*. Note that a *selected next-hop* may be different from an *original next-hop* for the same prefix as long as aggregated FIB has exactly the same forwarding behaviors as the original one.

4. *FIB status*, denoted by *F(n)*. Indicates whether the prefix and its selected next-hop should be placed in the FIB or not after FIB aggregation. *F(n)* can be equal to *IN_FIB* or *NON_FIB*. All routes with the status *F(n)* equal to *IN_FIB* account for the entire aggregated FIB.

We illustrate the initial PT for the FIB from Table 3.1a in Figure 3.2(a). At this stage, the selected next-hop value and the FIB statuses of each node are unknown. In addition, the *original next-hops* for *FAKE* nodes are not defined yet. All these missing attributes are set within a single traversal that we call *Aggregation Round* - a depth-first traversal over the PT in a post-order manner. For the illustration, we divide *Aggregation Round* traversal into repeating top-down and bottom-up phases. During the top-down phases FAQS algorithm sets the *original next-hop* value of *FAKE* nodes equal to their parent's *original next-hop* value. During the bottom-up phases, FAQS algorithm sets the *selected next-hop* for each processed node and the *FIB statuses* of its existing child nodes.

The *FAQS* algorithm chooses a selected next-hop for a node according to the **rule A**:

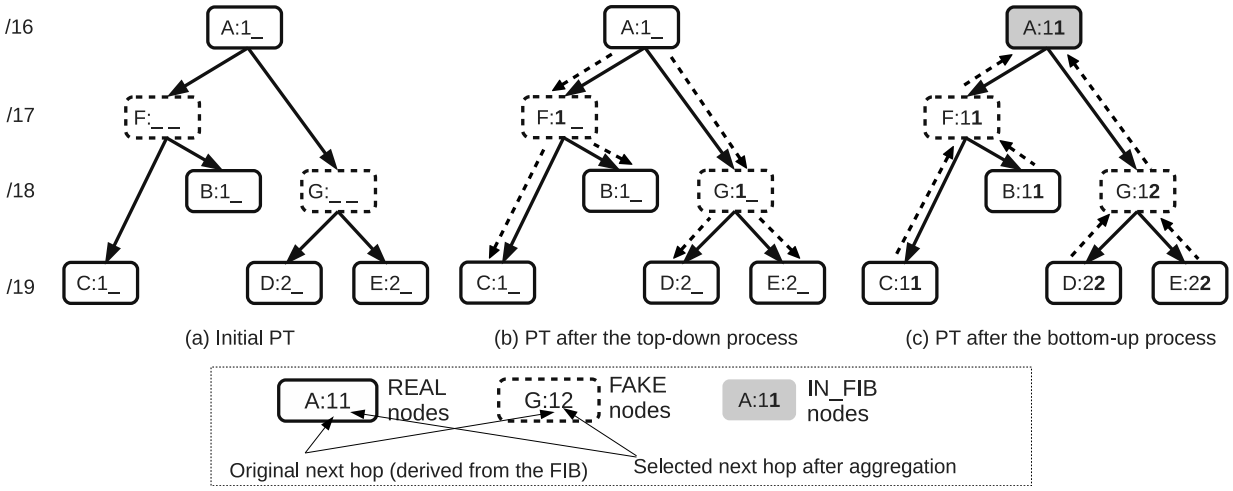


Figure 3.2: Static FIB aggregation of FIB from Table 3.1a.

I. If a node n **has both child nodes with prefix lengths more than *n*'s prefix length by one**, then n's *selected next-hop* value is equal to:

1. Its original next-hop value *O(n)*, if either left or right child nodes of n has the selected next-hop value equal to *O(n)*.
2. Left child's *selected next-hop* value in other cases.

II. In all other cases, the selected next-hop *S(n)* is set equal to node's original next-hop *O(n)*.
Note 1: All leaf nodes' *selected next-hop* values are equal to their *original next-hop* value.
Note 2: The root node with prefix *0/0* is always *REAL* and has a certain default *original next-hop* value, if its prefix does not exist in RIB.

Intuitively, the selected next-hop value of a node is equal to its original next-hop if its corresponding prefix is not fully overlapped by its children nodes' prefixes or its original next-hop is equal to one of its children nodes' selected next-hop values. Otherwise, when a node n's IP space is fully covered by its children's prefixes, we set n's original next-hop value to its left child's selected next-hop value. This rule allows *FAQS* to preserve forwarding equivalence of the resulting FIB since the next-hop value of a prefix changes only when (a) it was not a potential Longest Prefix Match in the original FIB table (i.e., its prefix space was fully covered by other more specific prefixes); (b) the new next-hop value of a prefix p is equal to a next-hop value one of p's sub-prefixes.

Label	Prefix	next-hop
A	141.92.0.0/16	1
G	141.92.192.0/18	2

Table 3.2: FIB entries after aggregation by FAQS.

After a node's *selected next-hop* value is set, *FAQS* makes decision on the *FIB status* of each of its existing child nodes according to the **rule B**:

> If the left (right) child of a node *n* has a *selected next-hop* value different from *n*'s selected next-hop value *S(n)*, then set the status of the left (right) child to *IN_FIB*, otherwise to *NON_FIB*.
>
> *Note:* the root node is always *IN_FIB*.

In other words, a node will be installed in the FIB if its selected next-hop is different from its parent's selected next-hop.

Intuitively, we start aggregation from the leaf prefixes and recursively assign selected next-hops based on their original next-hops. When a child's selected next-hop is the same as its parent's, the child's prefix and selected next-hop can be excluded from the aggregated FIB. The process stops at the root node, which is always *IN_FIB*. The resultant aggregated FIB will have exactly the same forwarding behaviors as the original one.

The illustration of the top-down and bottom-up phases of the initial aggregation are given in Figures 3.2(b) and 3.2(c). Post-order traversal begins with the leftmost node (node C) in the PT. While reaching node C, during the top-down phase, the *original next-hop* value of the *FAKE* node F is set to 1, the *original next-hop* value of F's closest $REAL$ parent A.

Node C has no children nodes, so its *selected next-hop* value is set to its original value 1. Similarly, the *selected next-hop* is set for all other leaf nodes. According to the ***rule A***, because the difference between the prefix lengths of the nodes F and C is more than 1, D's selected next-hop value is equal to its original next-hop value. In contrast, the difference of the prefix lengths of the node G and its children D and E is equal to one. Because the *selected next-hop* of D and E is different from G's *original next-hop*, G's selected next-hop is set to 2. Finally, A's *selected next-hop* is equal to 1 (similarly to the node F). Regarding *FIB status*, according to the ***rule B***, only nodes A and G will be installed in FIB, since A is the root node and its selected next-hop is not equal to G's selected next-hop. The resulting aggregated FIB is presented in Table 3.2.

Compared with the state-of-the-art *ORTC*-based *FIFA* algorithms [48], *FAQS* algorithm requires only one traversal for the static FIB aggregation. The worst-case complexity of the initial FIB aggregation is equal to $O(k * N)$, where:

- k is the maximum prefix length in the FIB. In real routing tables, the length rarely exceeds 24;
- N is the number of nodes in a PT[1].

FIFA requires two rounds: first, a post-order traversal in order to merge next-hops for each node, second, a pre-order traversal to choose a selected next-hop from the merged next-hop set and to set the *FIB status*. When aggregating real routing tables, the merged next-hop array in *ORTC* algorithm may contain tens of elements. Moreover, each node of the PT constructed by the *FAQS* algorithm has only half of the attributes, used in *FIFA* algorithms. This fact also adds to the reduction of memory usage and the number of memory access times, as the number of next-hops in merged next-hop arrays can surpass 70 (see Section 3.4 for the evaluation). The complexity of FIFA algorithms is $O(k * n * m)$, where m is the maximum number of elements in the merged next-hop array.

3.3.2 Incremental FIB update handling

Through BGP updates, ASes exchange routes between each other. During spikes, the number of updates reaches several thousand. Thus it is essential to efficiently handle such updates. BGP update handling should preserve the *Forwarding Equivalence* between the aggregated and original FIB tables.

BGP updates to a FIB consist of two categories:

1. Route announcements, including new routes and route changes;
2. Route withdrawals.

In the following, we describe how *FAQS* algorithm handles both categories.

• **Route announcements:** If the announced route is a new route, *FAQS* algorithm generates a *REAL* node with the corresponding original next-hop in the PT; if it is a route update, it simply changes the original next-hop value accordingly to the value propagated from the control plane. In order to maintain a good aggregation ratio and forwarding correctness, the aggregated FIB needs to be re-aggregated. In *FAQS*, two

[1] $N = 2 * p - 1$ in the worst case, where p is the number of prefixes in the FIB.

portions of the PT may be affected: the subtree rooted at the updated node and the ancestors upon it. More specifically, the original next-hop, the selected next-hop and the FIB status of each node under the sub-tree need to be checked and updated if necessary. The process is similar to the procedure of the static FIB aggregation for the entire PT. Also, the selected next-hop and the FIB status of each ancestor need to be checked and refreshed if necessary to maintain forwarding correctness. The procedure seems to be tedious, however, we leverage the following three crucial *optimization techniques* to greatly reduce the overall time costs and memory access times.

1. When adding a *REAL* node or updating a *FAKE* node, if the original next-hop of this node's parent *O(n.parent)* is same as the new next-hop of the updated node *O(n)*, then the top-down process can immediately terminate, since a parent's original next-hop in the subtree rooted at *n* does not change.

2. When updating the subtree, if the node type *T(n)* is *REAL*, then the top-down process can stop on the current branch because the original next-hop of that node does not change. Similarly to the first statement, the correctness is guaranteed because all *FAKE* nodes' original next-hops are derived from their nearest *REAL* ancestor's original next-hop. When the updated node is above this *REAL* node *n*, all original next-hops under *n* will not change.

3. During the period of updating the ancestors, if the newly selected next-hop of an ancestor *n* is the same as the old one before the update, then the bottom-up traversal can stop. Since update only happens on one branch and a parent's selected next-hop is determined by its children's selected next-hop, the preservation of a selected next-hop of a node *n* guarantees the invariance of all nodes above it.

Algorithms 4, 5 and 6 (see Appendix A) illustrate the whole process of incremental update handling. Figure 3.3 demonstrates an example of a route update with a new next-hop, where the second and third optimization techniques are applied. In the example, the node D has an update with the new next-hop 3. First, the original next-hop changes to 3 and other fields are freed; then the update-tree process stops when encountering a *REAL* node G. After that, the update-ancestor process stops when the same selected next-hop 1 is discovered at node B. As a result, we can observe that only a small portion of the PT has been traversed to incrementally handle the update.

In comparison with the *FAQS* algorithm, the ORTC-based *FIFA* algorithm [48] requires three steps for the prefix update handling. First, it is the bottom-up post-order traversal in order to update the merged next-hop sets. During the second step, the *FIFA* algorithm updates merged next-hop sets for each ancestor above the updated node. The last step is the post-order traversal of the PT, where *FIFA* selects the next-hop from the merged next-hop array and sets the FIB statuses as in the static aggregation algorithm. Overall, FIFA achieves the optimal aggregation ratio at the cost of a greater number of memory accesses during the

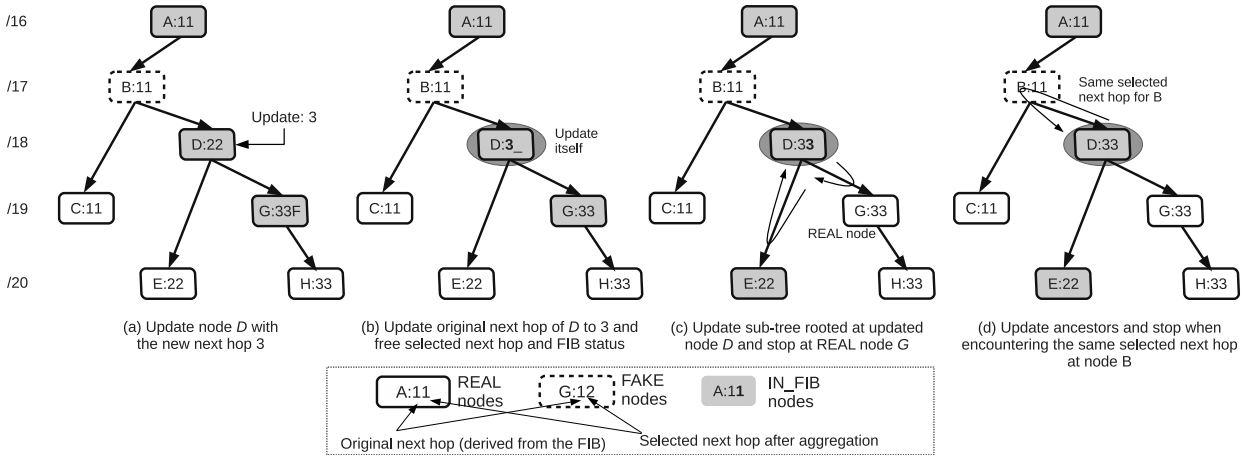


Figure 3.3: Incremental FIB update handling by FAQS.

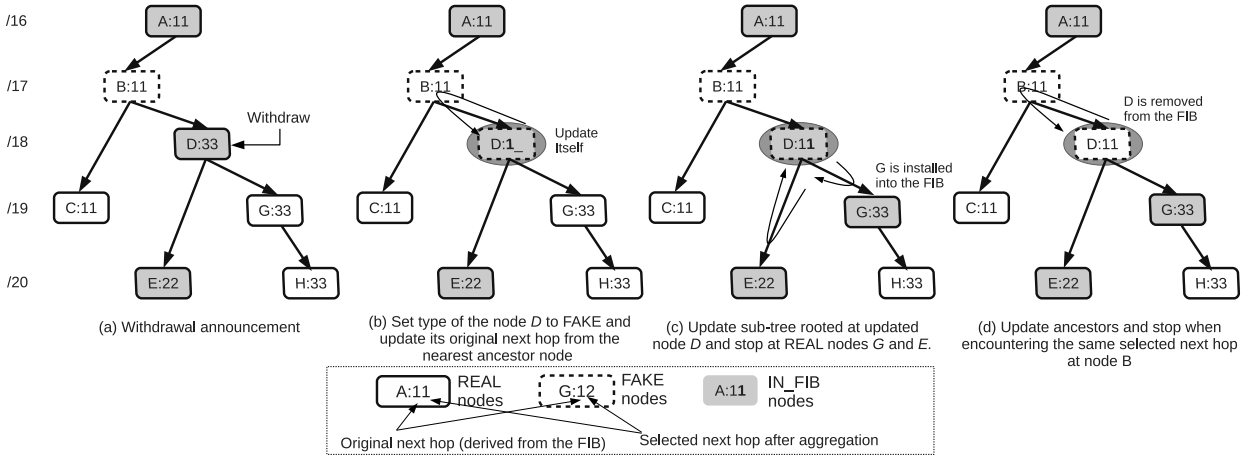


Figure 3.4: Prefix withdrawal handling by FAQS.

optimization of the updated PT. Later in Section 3.4 we show that this causes significant delays in processing FIB updates if compared to the *FAQS* algorithm.

• **Route withdrawals:** The *FAQS* algorithm handles the prefix withdrawals within two steps:

1. *Node removal.* First, *FAQS* looks up the corresponding *REAL* node from the PT. If the node is found, then *FAQS* checks if it is removable. A removable node refers to a node, which will not affect the PT structure after its deletion. In such a case, *FAQS* deletes the node and reorganizes the pointers of its parent and child. Otherwise, if the node is not removable, *FAQS* changes its type to *FAKE* and frees the values of the original next-hop, the selected next-hop, and the FIB status.

2. *PT update.* Starting from the parent node of the deleted or updated node, the incremental update process will be the same as the case of route announcements. First, *FAQS* does a top-down update of the original next-hops of nodes on the subtree; next, it bottom-up updates the values of the selected next-hops and the FIB status of each node all the way to the point where a newly selected next-hop does not change. The three optimization techniques used in route announcements apply here as well.

We illustrate prefix withdrawal operation in FAQS in Figure 3.4. In the example, the prefix associated with the node *D* should be removed. First, FAQS sets the type of the node *D* to *FAKE* and updates its next-hop with the nearest ancestor's original next-hop value. Since the node *D* has two children nodes, it is an ancillary node and can not be removed from PT. Next, *FAQS* updates the sub-tree rooted at *D* and stops at *REAL* nodes *E* and *G*. During this short traversal, *D* sets its selected next-hop to 1 (since the prefix length difference between *D* and *E* is 2) and *G* is pushed into the FIB. Finally, *FAQS* updates performs a bottom-up traversal and stops at *B*, since *B*'s next-hop does not change. Because *B*'s child node *D* has the same selected next-hop, *D* is removed from the FIB.

Similarly to update handling, *FAQS* requires fewer memory accesses to process route withdrawals. As route withdrawals are handled identically to route updates, both operations have the same worst-case complexity, $O(k * N)$, where k is the longest prefix in the table, and N in the number of nodes in a PT. However, in reality, the number of accesses is much less, thanks to the *optimization techniques*, namely, terminating the top-down traversal on *REAL* nodes and the bottom-up traversal on the nodes that do not change their selected next-hop.

3.4 Evaluation of FAQS

We used realistic IPv4 and IPv6 routing tables from 2011 to 2016 in Route Views project [5] for the evaluation. We collected several baseline routing tables on 01/01/2011 for both IPv4 and IPv6 and applied all following updates to obtain the aggregation results. We use AS neighbors as the next-hops for FIB tables, because local FIB interface information is not available in the dataset. Normally, the number of interfaces in a FIB is much less than the number of its neighbors. Thus our results underestimate the real FIB aggregation effects. We verified the forwarding equivalence of the aggregated and original tables with our designed tool *VeriTable* [32]. We briefly describe VeriTable in the next subsection. We ran our experiment on an Intel Xeon Processor E5-2603 v3 1.60GHz machine.

We compared our *FAQS* algorithm against the optimal ORTC-based *FIFA-S* [48] aggregation algorithm. Unlike *FIFA-T*, a faster version of *FIFA* algorithms, *FIFA-S* has significantly smaller FIB bursts, which is

critical since writing operations on TCAM are slow [12].

We used the following metrics for our experiment:

1. *FIB size*: the total number of entries before and after aggregation. *Aggregation Ratio* is calculated as the ratio between the total number of the FIB entries after aggregation and before aggregation.
2. *FIB aggregation time*: the time spent handling all route updates by the aggregation algorithm (before pushing FIB changes into the data plane).
3. *Total number of FIB changes*: the total number of FIB changes that are pushed into the data plane by the aggregation module upon handling all route updates. One route update from the control plane may result in zero or more changes to the data plane FIB due to the incremental FIB aggregation process. If there is no aggregation, one route update corresponds to one FIB change.
4. *FIB burst*: the number of FIB changes caused by a single route update, either a route announcement or a withdrawal.
5. *Peak aggregation time cost*: the time spent to handle a routing announcement, that caused the heaviest FIB burst.

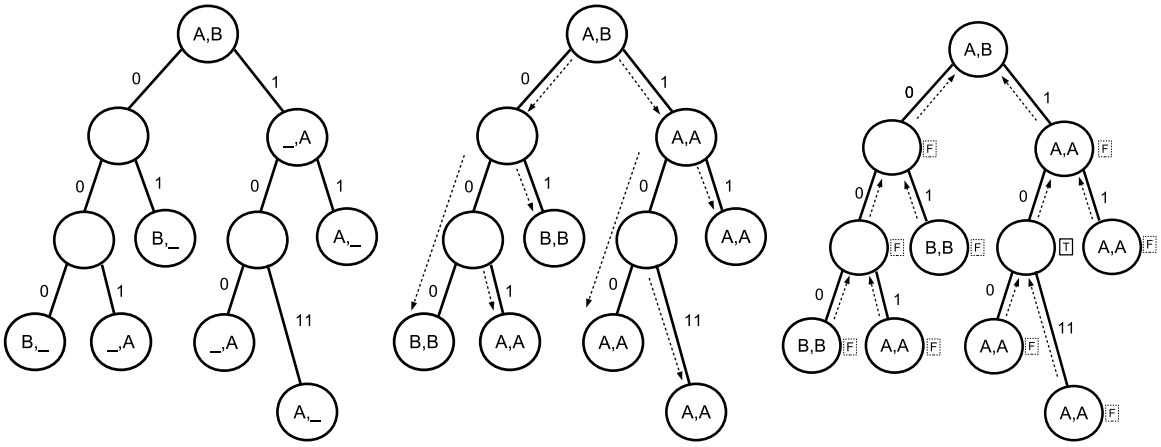
3.4.1 Verifying the Forwarding Equivalence with VeriTable

Verifying the equivalence of forwarding behavior of routing tables is essential for several cases. First, it is used in order to guarantee the correctness of the forwarding table in the data plane, derived from the control plane [19, 21, 68]. Second, network operators need to verify the consistency of the forwarding tables inside their networks [39, 40]. Finally, upon modification of the forwarding table due to FIB aggregation or caching techniques, the forwarding behavior of a router shall not change. Specifically for this case, we designed VeriTable [32], that is able to quickly verify the equivalence of multiple forwarding tables. The algorithm of VeriTable is built upon a theorem we prove below.

The formal definition of the Longest Prefix Matching and Forwarding Equivalence is given in Section 2.2 (Definition 1) and Section 3.1 (Definition 2) respectively.

In addition, we denote:

1. $LPM_{\omega}(T)$ as a Longest Prefix Match of an IP address ω in a table T.



(a) Initial Joint PT (b) Joint PT after the top-down process (c) Joint PT after bottom-up verification

Nodes with non-empty labels are derived from at least one of the comparable forwarding tables, while the rest of the nodes are auxiliary nodes helping to build the data structure. During the only post-order traversal over the tree, non-empty nodes are initialized with the next-hops with respect to each forwarding table. Finally, the next-hops are compared within each node, to find discrepancies between forwarding tables.

Figure 3.5: VeriTable algorithm.

2. $N_T(p)$ as a next-hop for a prefix p in a table T.

Theorem 1. *Let T to be a joint forwarding table, built by merging individual forwarding tables $T_1, T_2, ..., T_m$. Assume that $p = LPM_\omega(T)$, then we can prove that $\forall\omega = \{0,1\}^n$, $LPM_\omega(T_i) = LPM_p(T_i)$, where n is the length of an IP address and $i = 1, 2, ..., m$.*

Proof. Let a prefix p be $p_1p_2..p_l$ ($l \leq n$), and ω be $\omega_1\omega_2..\omega_n$. Suppose, $p = LPM_\omega(T)$, then $\omega = p_1p_2...p_l\omega_{l+1}..\omega_n$. We prove the theorem using contradiction. Suppose, $LPM_\omega(T_i) \neq LPM_{p_1p_2..p_l}(T_i)$, namely, $LPM_{p_1p_2...p_l\omega_l\omega_{l+1}..\omega_n}(T_i) \neq LPM_{p_1p_2..p_l}(T_i)$. Then, according to the Definition 2, there exists a different prefix p' in the forwarding table T_i, such as its length $l_{p'} > l_p$, and p' is a prefix for w. But then, p' exists in T. Thus, p can not be a Longest Prefix Match for ω in T (see Definition 2), which is contradictory to the initial assumption of this theorem, that $p = LPM_\omega(T)$. □

We derive VeriTable property based on the Theorem 1: comparing next-hops for each ω in $T_1, T_2, ..., T_m$ is equivalent to comparing next-hops for each p in $T_1, T_2, ..., T_m$. Thus the new definition of *Forwarding Equivalence*:

Definition 3. The forwarding tables $T_1, T_2, ..., T_m$ are *Forwarding Equivalent*, if and only if, $\forall\omega = \{0,1\}^n$, $\forall p = LPM_\omega(T)$, where T is the union of $T_1, T_2, ..., T_m$, $N_{T_1}(LPM_p(T_1)) = N_{T_2}(LPM_p(T_2)) = ... =$

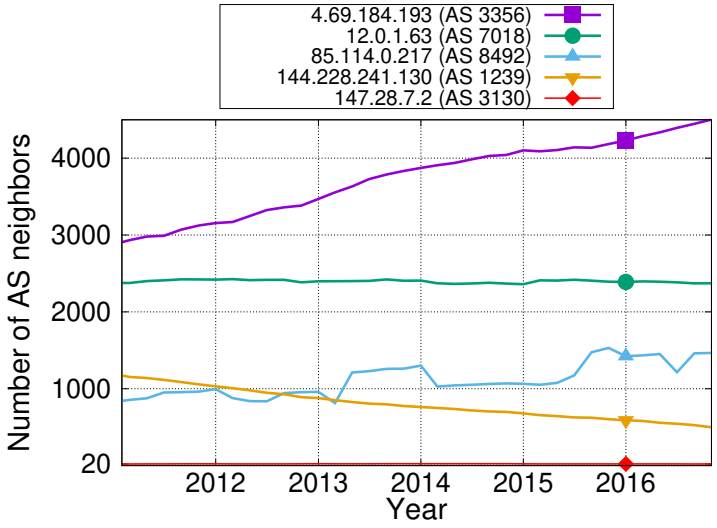


Figure 3.6: AS neighbor statistics.

$N_{T_m}(LPM_p(T_m))$.

Based on this new definition, we designed an algorithm, that builds a joint forwarding table out of the comparable tables (using the PATRICIA Tree data structure), identifies the next-hop values of LPM prefixes in each of the comparable tables and compares those values in order to find discrepancies. The pseudo-code of the algorithm is given in Appendix E. The efficiency of VeriTable was shown in [32].

3.4.2 IPv4 results

We use five routing tables from different ASes to demonstrate the dependency of aggregation performance on the number of neighbors (i.e. the number of possible next-hops). The number of next-hops ranges from 21 to 4500. To illustrate the extreme cases, when an AS has the maximum or the minimum number of neighbors, we took AS 3356 (hop IP address *4.69.184.193*) and AS 3130 (hop IP address *147.28.7.2*). AS 3356 had almost 3000 neighbors at the beginning of 2011 and more than 4500 neighbors on 12/31/2016. For AS 3130 these numbers are 22 and 21 respectively. The number of neighbors was calculated according to the routing tables of those ASes. See Figure 3.6 for the neighborhood statistics of each AS whose IPv4 data we used in this evaluation.

We begin with AS 3356. There are more than 426 million route updates to be handled for the 6-year period. Figure 3.7(a) shows the size of FIB aggregated by *FAQS* and *FIFA-S* algorithms, as well as the size of the

AS	n_u	p_{avg}	Algorithms	r	n_c/n_u	t_{aggr}	t_{peak}	$n_b = 0$	$n_b = 1$	$n_b \leq 30$	b_{max}	m
3356	426551755	3746	FAQS	0.43	1.27	0.94μs	3.09ms	12.75%	56.21%	99.97%	1443	175MB
			FIFA	0.37	1.84	2.38μs	10.29ms	12.85%	64.23%	99.98%	1496	228MB
7018	782293331	2397	FAQS	0.41	1.21	0.94μs	3.09ms	16.43%	61.71%	99.99%	1854	175MB
			FIFA	0.34	1.78	2.06μs	8.39ms	16.38%	54.57%	99.97%	1892	229MB
8492	1037150247	1126	FAQS	0.39	1.37	0.93μs	3.27ms	6.05%	69.45%	99.97%	4268	178MB
			FIFA	0.32	1.90	2.33μs	12.14ms	6.40%	63.77%	99.97%	4657	233MB
1239	295214072	739	FAQS	0.42	1.28	1.09μs	3.56ms	13.18%	62.18%	99.98%	1585	175MB
			FIFA	0.36	1.91	2.69μs	12.68ms	13.38%	55.03%	99.97%	1952	229MB
3130	402445005	23	FAQS	0.27	1.23	0.95μs	3.26ms	14.69%	63.50%	99.98%	6464	174MB
			FIFA	0.20	1.68	2.04μs	16.02ms	14.91%	56.74%	99.98%	5524	228MB

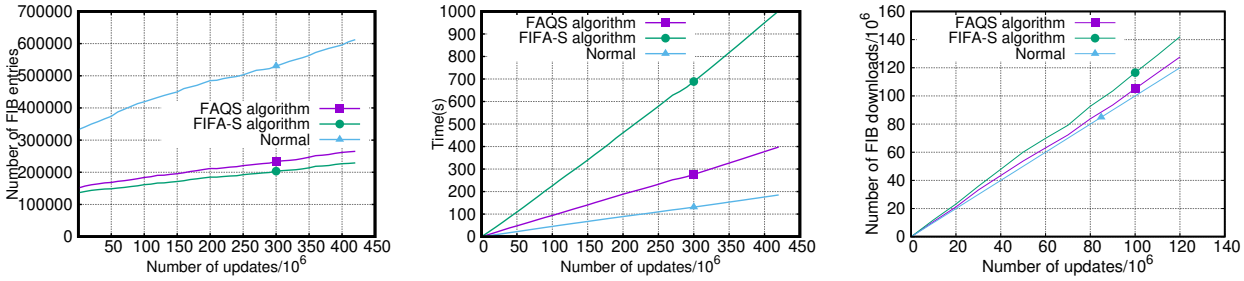
(a) IPv4 routing tables

AS	n_u	p_{avg}	Algorithms	r	n_c/n_u	t_{aggr}	t_{peak}	$n_b = 0$	$n_b = 1$	$n_b \leq 30$	b_{max}	m
6939	122903741	2725	FAQS	0.63	1.06	0.76μs	1.27ms	7.08%	84.19%	99.99%	181	11MB
			FIFA	0.61	1.18	1.33μs	2.97ms	7.09%	81.19%	99.98%	258	14MB
33437	33486605	7	FAQS	0.58	0.98	0.90μs	1.42ms	17.47%	73.68%	99.99%	2447	11MB
			FIFA	0.56	1.11	1.43μs	2.48ms	17.46%	68.33%	99.99%	2432	14MB

(b) IPv6 routing tables

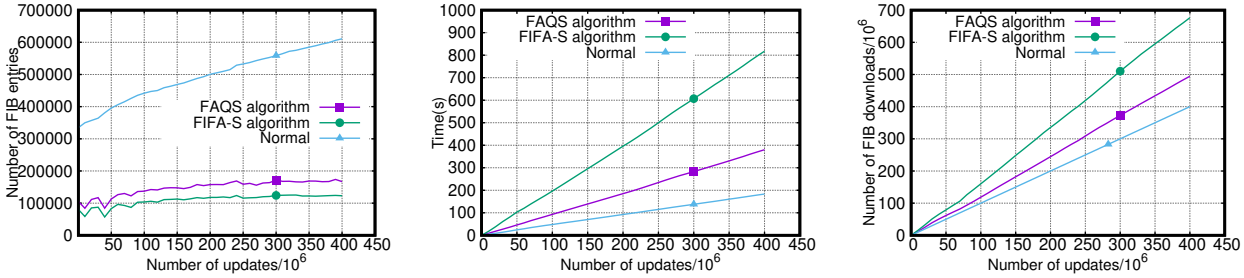
n_u - the number of FIB updates; p_{avg} - average peer number; r - aggregation ratio; n_c/n_u - the ratio between the number of FIB changes and FIB updates; t_{aggr} - average aggregation time per update; t_{peak} - peak aggregation time; n_b - percentage of updates with burst values 0, 1 and below 30; b_{max} - maximum burst value; m - memory consumption.

Table 3.3: Evaluation summary.



(a) FIB size (b) FIB aggregation time (c) FIB changes

Figure 3.7: FIB aggregation of IPv4 routing table (AS 3356).

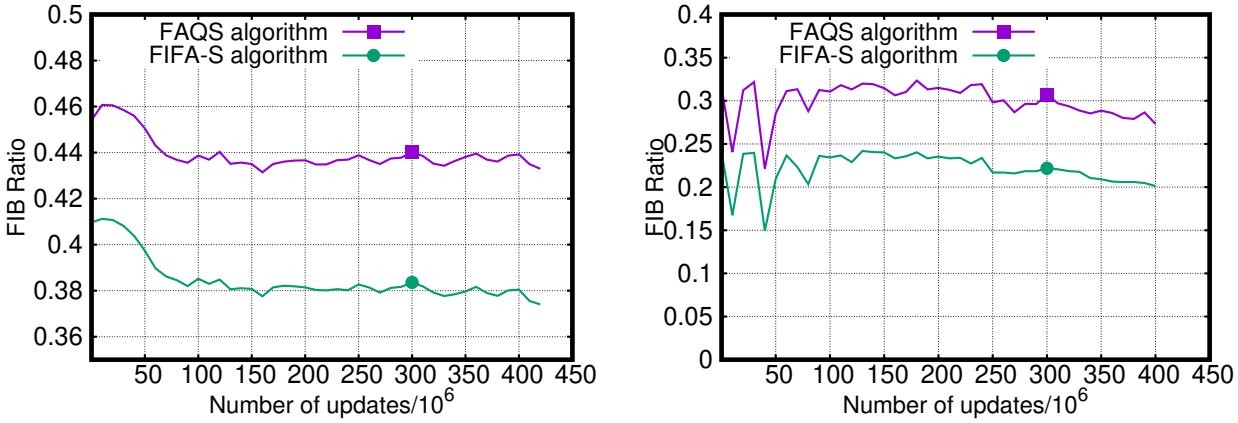


(a) FIB size (b) FIB aggregation time (c) FIB changes

Figure 3.8: FIB aggregation of IPv4 routing table (AS 3130).

original FIB during the 6-year period from 2011 to 2016. Figure 3.9(a) more clearly illustrates the margin between FIB ratio values of *FAQS* and *FIFA-S*. At the end of 2016, *FAQS* algorithm reduces the FIB by ***57%***. *FIFA-S* reduces the same FIB by ***63%***. The better aggregation ratio by *FIFA-S* algorithm is achieved by additional traversals. This results in higher aggregation costs, as illustrated in Figure 3.7(b). Aggregation time costs when using *FAQS* are more than twice less. On average, *FAQS* spends ***0.94***μs per one FIB update. For *FIFA-S* algorithm the average FIB update cost was ***2.38***μs. The peak aggregation time in *FAQS* is 3 times less than in *FIFA* (3.09ms vs 10.29ms).

The smaller number of FIB changes to the FIB, the better performance. Figure 3.7(c) shows that ***FAQS* algorithm generates 31% less number of FIB changes than that of *FIFA-S* algorithm** (543,309,259 vs 786,633,132). The average number of FIB changes per update is 1.27 for *FAQS* and 1.84 for *FIFA-S*. Both algorithms have similar distribution for the size of FIB bursts as shown in Table 3.3(a). The vast majority of FIB bursts (more than 99.97%) in both algorithms consist of 30 FIB changes and less. The largest FIB burst for *FAQS* is 1443, which is slightly smaller that *FIFA-S* (1496). Nonetheless, **the update handling**



(a) AS 3356

(b) AS 3310

Figure 3.9: FIB ratio, IPv4 FIB tables.

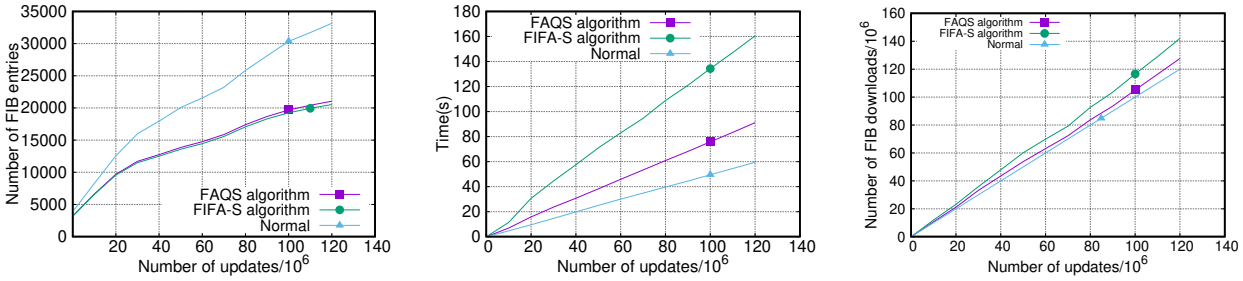
time cost for the largest burst in *FAQS* takes only 30% of running time of *FIFA-S*.

Since AS 3130 has a considerably smaller number of neighbors than AS 3356, it is expected that the aggregation ratio for its routing table is smaller. Indeed, Figure 3.9(b) shows that *FAQS* can reduce the FIB size by ***73%*** and *FIFA-S* by ***80%*** at most. The more compressed state of FIB may cause larger FIB bursts, as we can see from this experiment. In the case of AS 3130, the percentage of FIB bursts with less than 30 FIB updates is again negligible: ***99.98%*** for both *FAQS* and *FIFA-S*. The maximum FIB burst value for *FAQS* was ***6464***, while for *FIFA-S* it was equal to ***5524***. However, *FAQS*'s peak aggregation time cost was ***3.26 ms***, which is less than FIFA-S' (***16.02 ms***) by ***80%***. Figure 3.8 illustrates performance of *FAQS* and *FIFA-S* for the next-hop AS 3130.

Table 3.3(a) presents other evaluation results of FIB aggregation for five ASes. It is interesting to observe that a large percentage (6.05%-14.91%) of FIB updates result in zero FIB changes (column n_b=0). Overall, for IPv4 protocol, *FAQS* algorithm is more than two times faster than *FIFA-S* on average and is more than three times faster during peak FIB bursts.

3.4.3 IPv6 results

To the best of our knowledge, in this work, we for the first time evaluate IPv6 routing tables aggregation. We aggregated FIB tables from AS 6939 (hop IP address *2001:470:0:1a::1*) and AS 33437 (hop IP address *2001:4810::1*) with the maximum and the minimum (3501 and 7 respectively) number of neighbor ASes in our collected dataset. The total number of route updates to be handled is more than 122 million. Figure 3.10



(a) FIB size (b) FIB aggregation time (c) FIB downloads

Figure 3.10: FIB aggregation of IPv6 routing table (AS 6939).

shows the curves of FIB size, aggregation time and the total number of FIB changes. In Figure 3.10(a), we can observe that the size of IPv6 routing tables has increased dramatically since six years back when there were only less than 5,000 entries. At the end of 2016, it has been close to 35,000. Due to the small size, the aggregation ratios for both *FAQS* and *FIFA-S* are around 60%, which are not as good as IPv4. Since *FIFA-S* outputs the smallest aggregated FIB, *FAQS*'s aggregation ratio for IPv6 is close to optimal. Remarkably, the running time of *FAQS* is a much lower than *FIFA-S* (90s vs 160s in Figure 3.10(b)) while they have similar aggregation ratios, which again attributes to the one-time subtree traversal with three important optimization techniques for *FAQS* while *FIFA-S* uses two traversals. Table 3.3(b) demonstrates results for both AS6939 and AS33437. AS33437 has only 7 next-hops, thus the aggregation ratio is better (58% vs 56% for *FAQS* and *FIFA-S*, respectively) and the burst size is larger than the one in AS6939, because one update in AS33437 may affect a larger area of next-hops.

3.5 FAQS. Conclusion

FIB Aggregation with Quick Selections (FAQS) is a new FIB aggregation algorithm, that leverages compact data structures and three unique optimization techniques to quickly and incrementally select next-hops when handling route updates. As a result, *FAQS* can run up to 2.53 and 1.75 times faster for IPv4 and IPv6, respectively, than the optimal FIB aggregation algorithm while achieving near-optimal aggregation ratio. Meanwhile, it consumes much less memory and generates a significantly smaller number of FIB changes when carrying out frequent updates. The performance enhancement of the new algorithm addresses many concerns from ISPs regarding performance issues and enhances the probability to push FIB aggregation techniques further to the level of production adoption by the industry.

Chapter 4

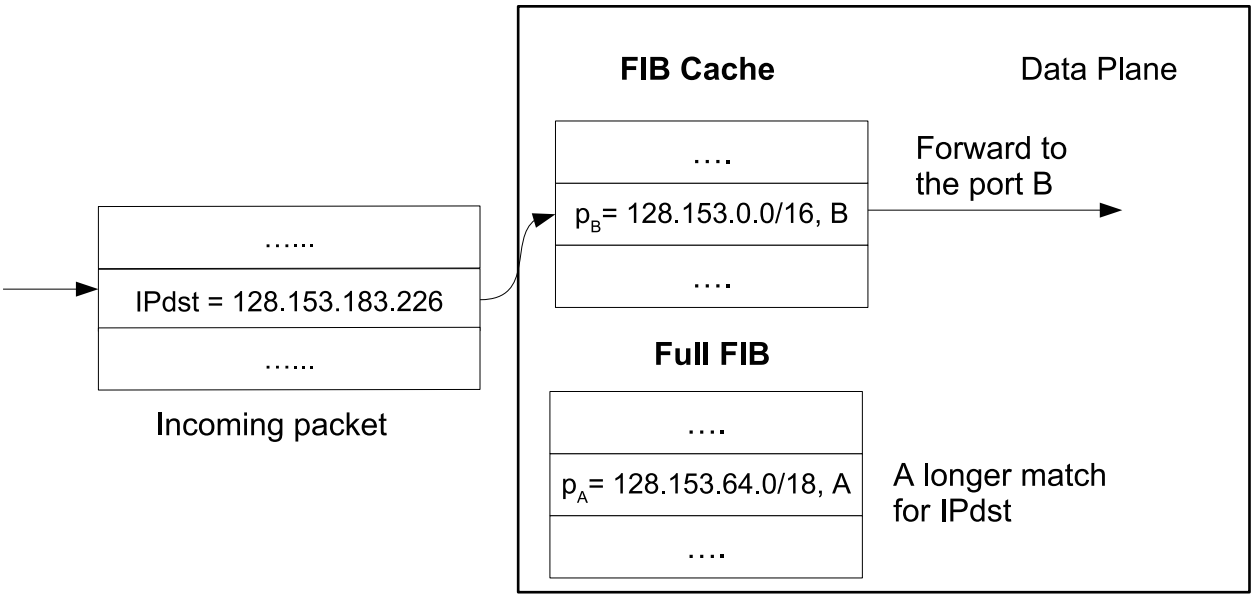
A Programmable FIB Caching Architecture

4.1 Introduction

FIB (or route) caching, i.e., storing popular IP prefixes in the expensive TCAM memory and the rest of it on a cheaper memory, leverages the fact that only a small part of IP prefixes carries the most of the traffic. In fact, according to the evaluation in [25], only 1.93% of FIB entries cover more than 99.5% of flows going through an ISP network. Moreover, according to Comcast, 72% of FIB entries do not carry traffic over a week [14]. The high skewness between the number of popular and unpopular routes was studied and confirmed in [41, 64]. According to these statistics, FIB caching may greatly offload the routing table in TCAM from unpopular routes, prolonging the lifetime of these expensive memory chips and reducing its operational costs.

However, FIB caching faces several challenges that impede its widespread deployment at an ISP level. First, it may result in latency issues when cache missing events occur, i.e., when an LPM lookup at TCAM fails and an additional lookup is performed on a slow memory. Moreover, simultaneous cache misses can lead to packet losses if a slow memory's buffers are full [41]. Second, to the best of our knowledge, there is a lack of an efficient cache victim selection procedure. LRU (Least Recently Used) policy, shown to be the most optimal in [25,41], is a software solution and can not be implemented for large prefix tables [75] on TCAM. Third, overlapping entries in a FIB cache and the secondary FIB might lead to the problem known as *cache hiding*.

Cache hiding is a natural problem that comes from dropping entries from a routing table that uses the LPM rule to find a match for an IP address. For example, suppose the secondary FIB contains a more specific



An IP packet was forwarded to the port B instead of the port A
due to the cache hiding error

Figure 4.1: Cache hiding example.

prefix $p_A = 128.153.64.0/18$ than a prefix $p_B = 128.153.0.0/16$ existing in a cache. The next-hop assigned to p_A in the secondary FIB is A, while IP packets with addresses matching p_B are forwarded to a next-hop B. In such case, a packet with IP destination address $128.153.183.226$ matching p_A will be forwarded to the next-hop B rather than A, since in the cached FIB its Longest Prefix Match is the prefix p_B. We illustrate this case in Figure 4.1. Cache hiding may be caused by:

1. Incorrect initial cache with less specific prefixes than in the secondary FIB on a slow memory;
2. A BGP prefix announcement, when a more specific prefix than another prefix existing in a cache is installed into the secondary FIB;
3. Similarly, BGP withdrawal might cause a situation when a cache contains less specific entries than in the secondary FIB.

When designing PFCA, we aimed to minimize cache-miss latencies, develop an efficient and fast cache victim selection procedure and avoid cache hiding problem. To this end, we leverage the emerging concept of the programmable data plane which we briefly describe next in this work.

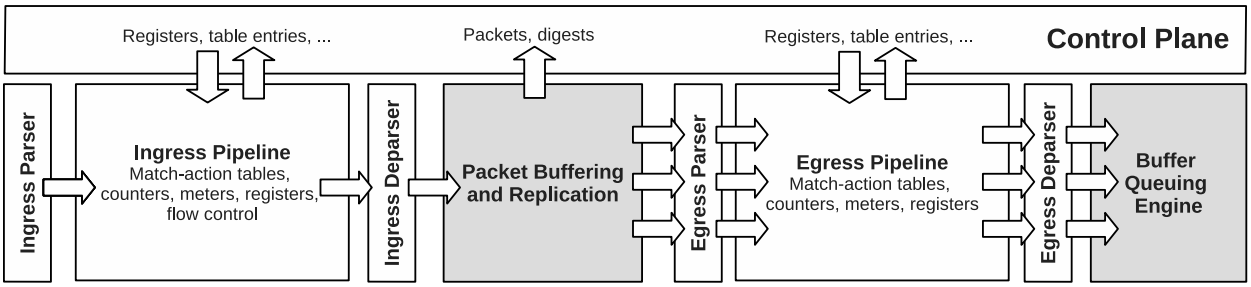


Figure 4.2: Portable Switch Architecture (PSA).

4.2 Programmable data plane

The numerous emerging heavy-workload IT applications require the underlying network to quickly adapt to their demands. Thus there is a need to build more programmable and less hardware-dependent networks. OpenFlow protocol [50] and Software-Defined Networking [42] in general are important tools for building programmable networks focusing on a programmable control plane. However, OpenFlow switches are constrained with a fixed data plane configuration. Differently, the switches standardized with the Portable Switch Architecture (PSA) [58] provide the data plane with a reconfigurable parser, forwarding (or match-action) tables, and ingress/egress pipelines. PSA switches can be fully programmed with P4 data programming language [13]. In addition, PSA switches support packet cloning, packet recirculation, shared metadata between different stages of the packet processing pipeline and provide access to data plane registers and hash calculations.

Figure 4.2 shows the pipeline of PSA. It consists of programmable blocks, such as ingress and egress pipelines, packet parsers and deparsers; hardware-constrained blocks responsible for packet replication and buffering. The ingress and egress pipelines may contain registers and match-action tables with packet counters and meters attached to each entry. An output port is assigned to a packet when its' headers leave the ingress pipeline. Alternatively, a packet can be cloned to multiple output ports or forwarded to the control plane. In addition, PSA data plane can be programmed to form packet digests for further analysis at the control plane.

In our work, we leverage both PSA and P4 to design an architecture for efficient FIB caching. In addition to our data plane program, we use API generated by P4 compiler to build the partial prototype of our FIB caching solution and simulate it in a P4 software switch. Note that the work on PSA compiler is still in progress, however, the design proposed in this work is compatible with the proposed PSA specification [58].

We call our new design a Programmable FIB Caching Architecture (PFCA). In PFCA, we adopt several

features of PSA, P4 and programming data planes in general. We enumerate those features below:

1. We take advantage of the programmability of match-action tables to design two different levels of a FIB cache in the data plane. In addition, the rest of a FIB on a slow memory unit stays in the data plane as well.

2. We leverage the programmability of the ingress and egress pipelines in PSA, which allows PFCA to process cache-miss packets entirely in the data plane. Packets that did not find a matching entry in Level-1 cache are directed to Level-2 cache and the slow FIB, located on the slower memory units.

3. Several of the targets for programmable data plane, like TOFINO switches, support parallel packet lookup against multiple match-action tables [7, 13]. In our design, packets that found no matching prefix in Level-1 cache are then matched against Level-2 cache and the slow FIB in parallel, in order to decrease cache-miss latency.

4. PFCA uses P4 meters, registers and hash functions to identify unpopular and popular routes, trigger cache replacement operations and conduct pipeline-based packet processing. More specifically, we detect the heavy flows and collect the light-weight flows for the future cache replacement ”on-fly”, without a need for iterating through large forwarding tables.

5. The programmability of PSA switches allows the network operator to tune a switch's parameters, such as cache installation thresholds, the cache sizes and light-weight flow filter size in order to adapt the switch to different traffic patterns.

4.3 Design

4.3.1 Overview

We illustrate the design of PFCA in Figure 4.3. Same as in a typical router, the control plane in PFCA is responsible for collecting and distributing routes, selecting best routes and installing them into the forwarding tables. In addition to that, the PFCA control plane contains the Route Manager module, which stores the full FIB, generates the set of non-overlapping routes and distributes them among the tables of the data plane. The data plane of PFCA is based on Portable Switch Architecture (PSA) and consists of a packet parser, ingress and egress pipelines, packet deparsers and buffers. We construct the ingress pipeline of the PFCA switch with three forwarding tables:

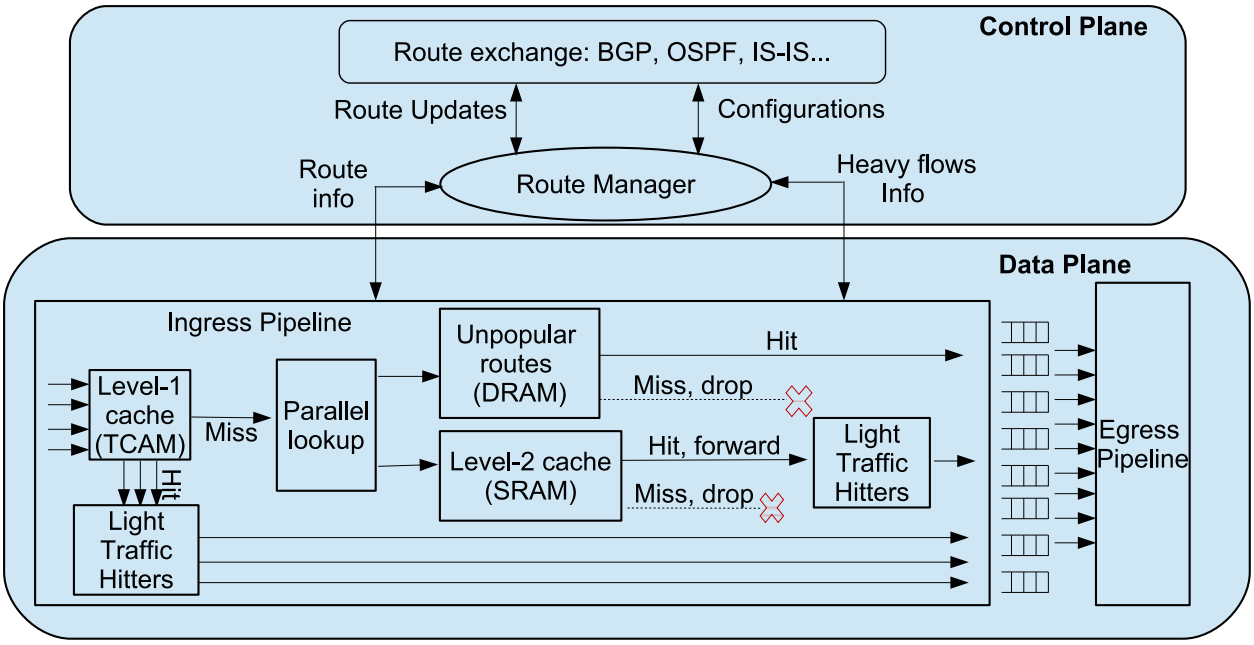


Figure 4.3: FIB caching architecture.

1. Level-1 cache stored in TCAM for the most popular prefixes;
2. Level-2 cache stored in SRAM for less popular prefixes;
3. The secondary FIB with unpopular prefixes in DRAM (a ”slow” FIB).

The essential part of our design is the *Light Traffic Hitters Detection* module, that is placed in the ingress pipeline and processes packets that were assigned a next-hop in Level-1 or Level-2 cache. The two-layer cache architecture with an SRAM chip in the Level-2 cache minimizes delays in case of Level-1 cache misses. To keep track of the most popular flows, every FIB entry in Level-1 cache, Level-2 cache, and the slow FIB is linked with a traffic popularity meter.

Overall, PFCA does not require specific hardware compared to standard programmable switches such as Tofino, where the ingress pipeline is built upon different types of memory, such as TCAM and SRAM [74]. The two-layer cache architecture with an SRAM chip in the Level-2 cache minimizes delays in case of Level-1 cache misses. To keep track of the most popular flows, every FIB entry in Level-1 cache, Level-2 cache and the slow FIB is linked with a traffic popularity meter.

PFCA's overall workflow is as follows: first, when a packet arrives at the data plane, the parser decouples the Ethernet and IP headers of a packet. Next, the packet's destination IP address is matched against Level-

1 cache, located on a TCAM chip. If there is a hit, the matched prefix is processed through the *Light Traffic Hitters Detection* module, that collects into its hash tables the least popular prefixes as cache victims. Subsequently, the packet's Ethernet frame header is rewritten, the packet is passed to the egress pipeline, re-assembled and forwarded to the corresponding next-hop. If there is a Level-1 cache miss, the packet is matched to both the Level-2 cache and DRAM table in parallel. The match on Level-2 cache leads to the same procedure as with a Level-1 cache hit, except that in this case, *Light Traffic Hitters Detection* operates with the prefixes from the Level-2 cache. For a match on DRAM, there is no further processing of a packet and it is immediately passed to the egress pipeline.

We leverage parallel table matching [7,73] of the programmable data plane in order to minimize latencies in case of a Level-2 cache miss. The overhead may be a concern here, but note that as we show in Section 4.6, only a tiny part of the traffic (less than 0.2%) is forwarded towards the second stage table matching. For instance, if the total traffic rate is 1 Tbps at a router, then Level-2 cache and DRAM only needs to handle less than 2 Gbps of the traffic, which is an easy task for a typical modern routing device. Therefore, compared with an existing router without caching capabilities, both hardware costs and energy consumption in the new solution will be significantly reduced.

Next, we present the detailed description of the Route Manager functionality and *Light Traffic Hitters Detection* module.

4.3.2 Route manager

The Route Manager (RM) belongs to the control plane and is responsible for several tasks in PFCA:

1. Control plane initialization;
2. Data plane initialization;
3. BGP update handling;
4. Popular routes installation.

In this section, we give a detailed description of each of these tasks.

Control plane initialization

RM receives the initial FIB entries from the control plane's BGP instance. Alternatively, the control plane may collect the network routes through OSPF, IS-IS or other network protocols; in this work, we assume that the control plane uses BGP. To avoid cache hiding of a prefix (see Section 4.1), RM converts the original prefixes into a set of non-overlapping prefixes. We call this procedure as "*prefix extension*". Prefix extension is performed by adding the original prefixes into the full binary tree with the following properties:

1. Each node in the tree may have zero or two children.
2. The root node of the tree corresponds to the root prefix 0/0.
3. The left (right) child of a node corresponds to a more specific prefix with an appended *0* (*1*) bit.

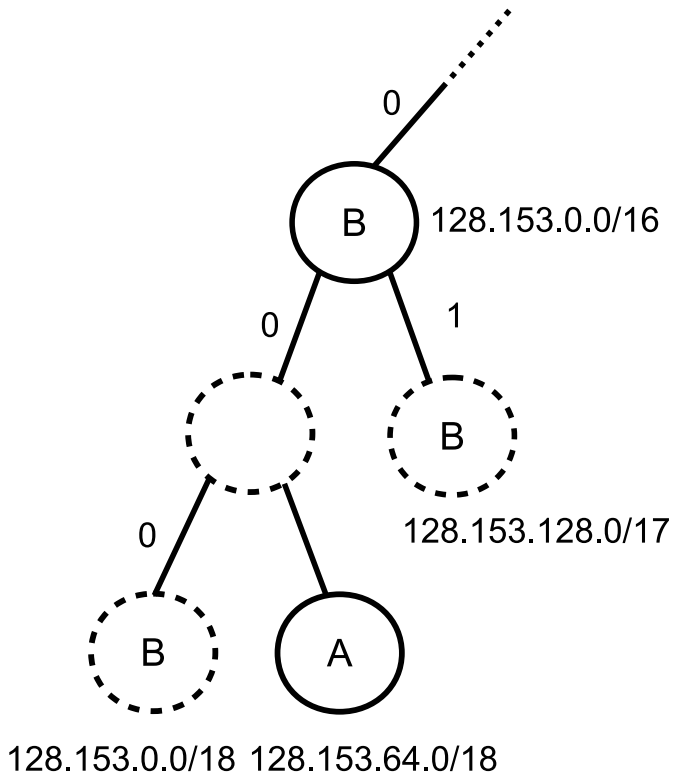
To add a new prefix p with a length l to the full binary tree, RM traverses the tree from the root, using the path defined by the bits of p. If the path ends at a level *l'*, RM will generate *l-l'* sibling nodes to ensure that the new set of leaf nodes will not contain any overlapping prefixes. The nodes generated during prefix extension inherit the next-hops of their closest parent nodes that relate to an original prefix.

Alternatively, we could generate non-overlapping prefixes on-fly, when moving less specific prefixes into the higher-lever memory. However, with such an approach, a "hot" entry migration to a cache might be delayed by prefix extensions due to costly computations at the control plane (see Section 4.5 for complexity analysis). To avoid these delays, PFCA performs prefix extensions pro-actively upon BGP updates.

After all original prefixes are added to the full binary tree, RM collects the set of non-overlapping prefixes using the leaf nodes of the tree. Each leaf node has four parameters:

1. The value of the prefix associated with a node;
2. The corresponding next-hop;
3. Cache flag, indicating the current location of the prefix in the data plane. Possible values are *0* (DRAM), *1* (Level-1 cache on TCAM) and *2* (Level-2 cache on SRAM).
4. REAL/FAKE flag, indicating if this node and its prefix was generated during prefix extension.

The new set has an equivalent forwarding behavior as the original routing table. Meanwhile, the size of the non-overlapping set is larger than the size of the original routing table. For example, an FIB containing



Prefix	Next hop
128.153.0.0/16	B
128.153.64.0/18	A

Original prefixes

Prefix	Next hop
128.153.128.0/17	B
128.153.0.0/18	B
128.153.64.0/18	A

Non-overlapping prefixes

The dashed nodes are generated when adding the prefix *128.153.64.0/18* to the tree.

Figure 4.4: Prefix extension in PFCA.

599,298 prefixes generates 844,108 non-overlapping prefixes [32]. However, the increase in size of the full FIB is not a concern here, as FIB caching dramatically reduces the number of entries in the expensive TCAM memory, as we show in Section 4.6. In addition, since we use only non-overlapping prefixes, the LPM operation on TCAM does not require strict ordering of the table's entries by their prefix length.

We illustrate prefix extension in Figure 4.4, where we use prefixes from the cache hiding example given in Figure 4.1.

Data plane initialization

Initially, the cache flag values for every node in a tree is set 0 and the corresponding prefixes are installed only into the FIB table on DRAM. There are several ways to fill up the initial caches. One way is simply a traffic-driven initialization ("cold start"), based on real-time pass-through traffic. More specifically, as network traffic goes through the switch, popular prefixes are being identified and migrated to the Level-2 cache table, located on an SRAM chip. Eventually, the most popular entries in the Level-2 cache are migrated to the Level-1 cache. As RM performs the migration task, it updates the flags of corresponding leaf nodes of its prefix binary tree.

Another way to initialize FIB caches is by analyzing the traffic history to identify the most popular routes and install them into the cache before operating the router. Alternatively, in [47], authors select the less specific routes for the initial cache. In this work, we use "cold start", as it results in a more stable cache than the other approaches.

BGP update handling

BGP updates (i.e. route announcements and withdrawals) are notified to RM by the control plane's BGP daemon. Since caches and a full FIB in the data plane contain extended non-overlapping prefixes, a single BGP update may result in multiple changes. In this subsection we describe how PFCA handles each type of a BGP update.

• For an **announcement of a new next-hop for an existing prefix**, RM first updates its own version of the full FIB (i.e., the prefix binary tree). If the updated prefix p is located on a leaf node, then RM needs to push the update of p into the data plane. Otherwise, if the updated prefix p is located on an internal node, then RM performs a tree traversal in order to update all *FAKE* nodes that inherited their next-hop value from p. For each traversed leaf node, RM pushes the updates into the forwarding tables, according to the cache flag

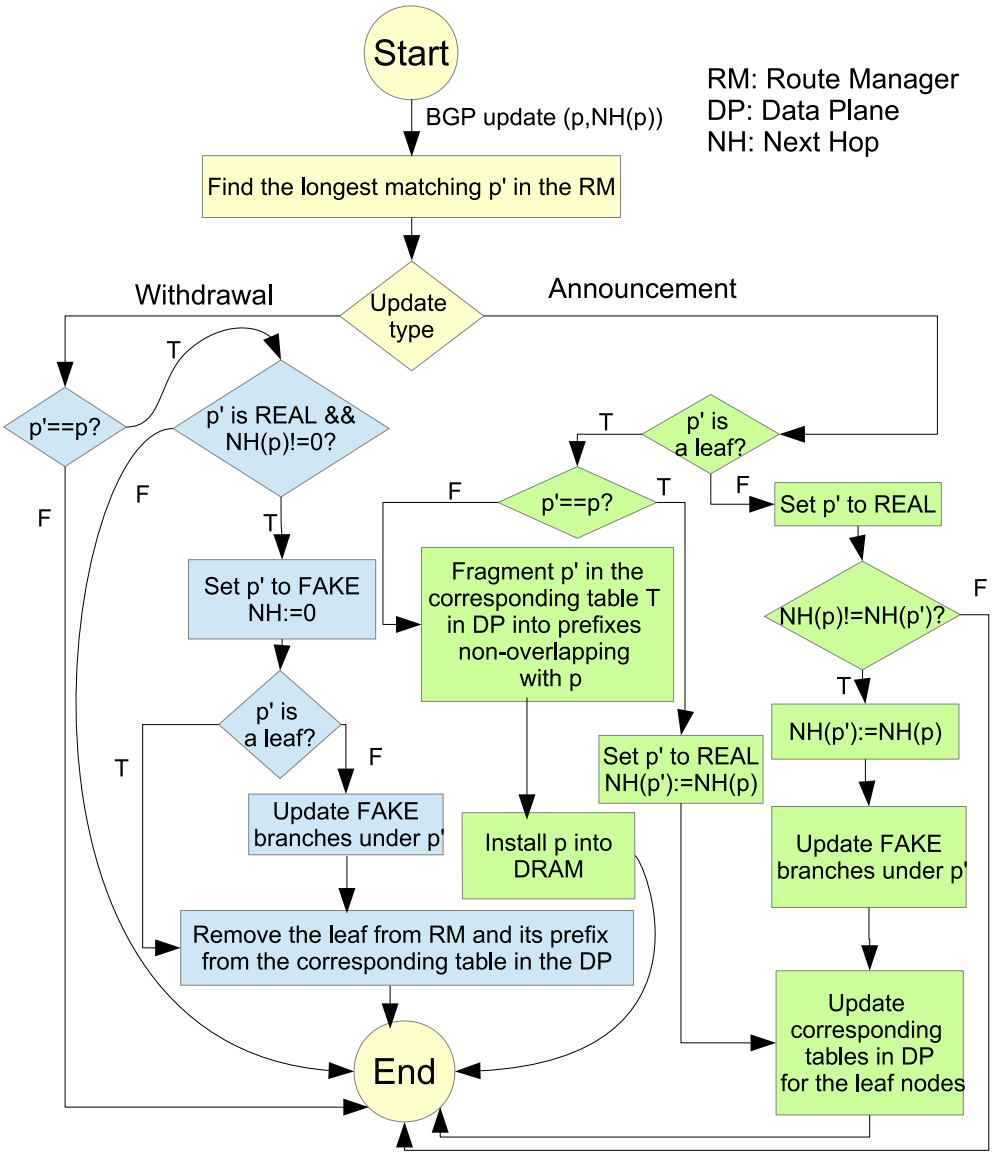


Figure 4.5: BGP updates processing by RM.

of that node.

• **Announcement of a new prefix** is less trivial since it requires additional check if such update introduces new overlapping prefixes in the forwarding tables of the data plane, which can lead to the cache hiding problem. By default, the route manager needs to install the new prefix into the slowest memory. If a new prefix p is more specific than an existing prefix p', RM fragments p' into a set of non-overlapping prefixes that includes the prefix p. This is done by installing the node for p and generating sibling nodes for each level of the tree between p and p'. These nodes will inherit the next-hop value of p' and will comprise the set of fragments of p'. After, (1) RM installs those fragments into the table where p' was located; (2) p is being installed into DRAM.

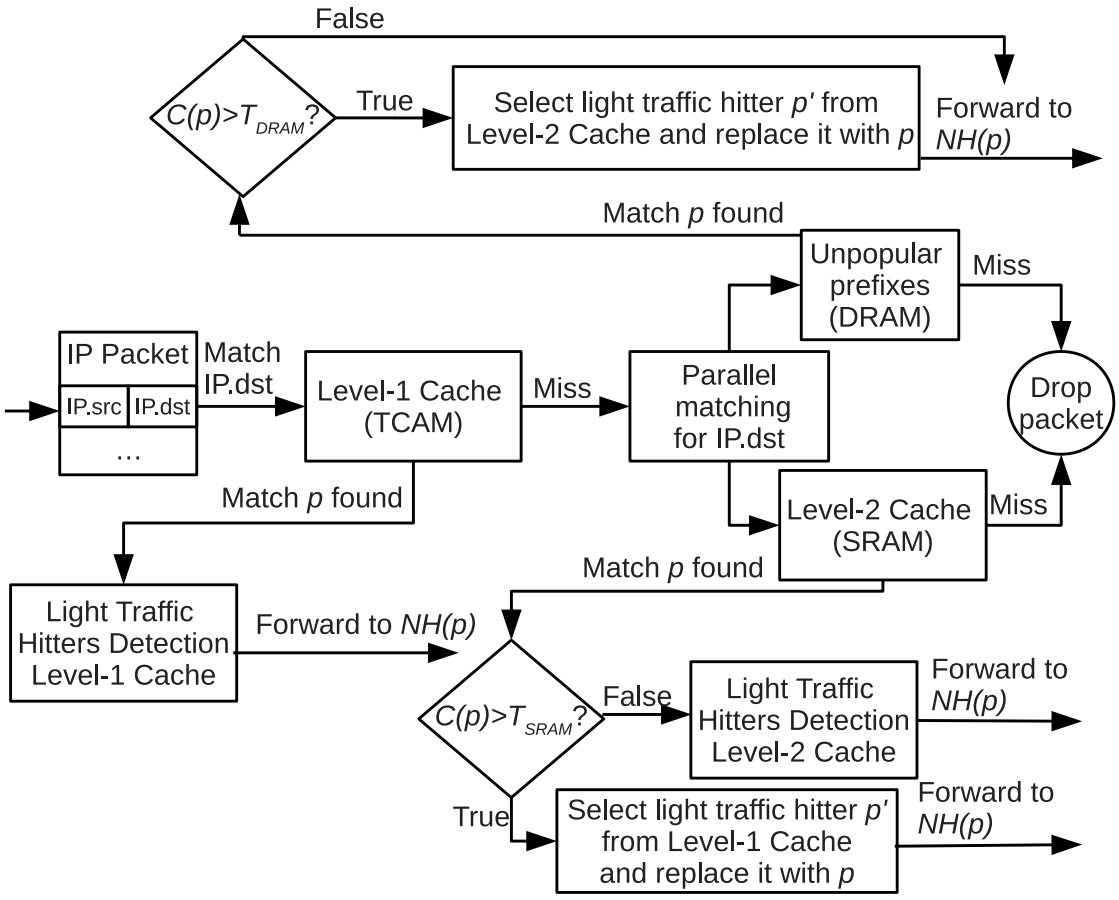
• For a **prefix withdrawal**, RM checks if such prefix *p* or its fragmented prefixes exist in the data plane and the hash tables of the *Light Traffic Hitters Detection* module and removes the prefix(es) from there if so. More specifically, RM traverses the branch of its prefix binary tree rooted at *p*, and checks the flags of the *FAKE* leaf nodes that inherited their next-hop from *p* and performs prefix deletions from the forwarding tables, respectively to the values of the flags.

Although PFCA allows multiple changes in the forwarding tables upon a single BGP update, most of the BGP updates do not affect the popular routes in both caches, as we show in Section 4.6. Therefore, a great advantage of using PFCA is that it can avoid massive expensive writing operations on TCAM and SRAM, and thus boost the network throughput.

We present the graph that illustrates a BGP update process in Figure 4.5. Note that the root node in the binary tree at RM is assigned with a default next-hop 0.

Popular routes installation

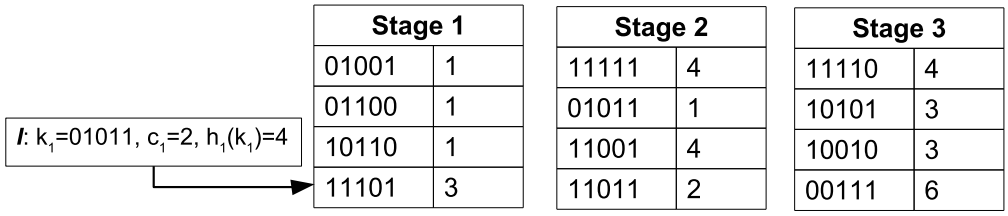
In our design, RM performs prefix migration to a faster cache table once a prefix installed in the Level-2 cache in SRAM or the slow FIB in DRAM reaches a certain number of matches over a period of time, e.g., 100 hits/s. A threshold can be configured depending on several factors, such as the volume of the traffic and the available memory on TCAM/SRAM. After the prefix is migrated, its counter is set to 0. In case a faster cache table is full, before installing a new entry, RM randomly picks a victim cache entry from the *Light Traffic Hitters Detection* module and evicts that entry back to the slower memory (see Figure 4.6 for the data plane workflow). We describe the cache victim selection process in the next subsection.



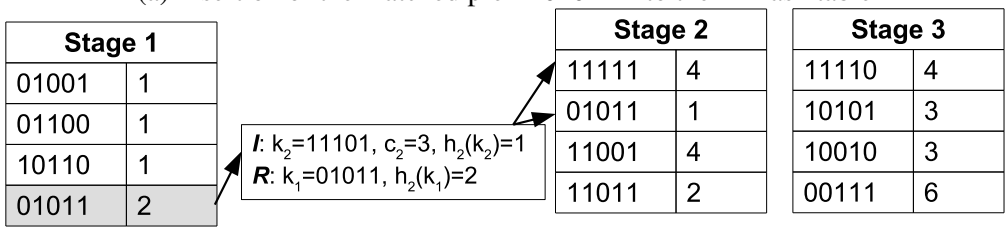
p: Prefix $NH(p)$: Next-hop for p

$C(p)$: Counter for prefix p T: Threshold

Figure 4.6: Data plane workflow

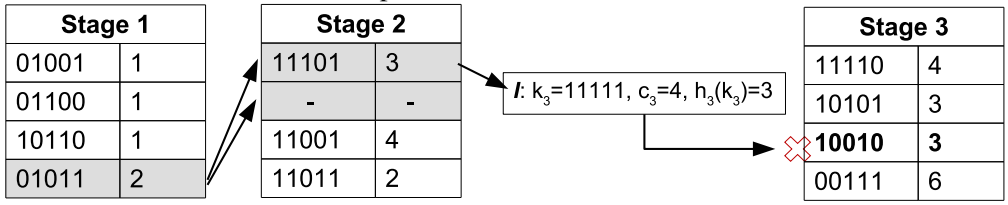


(a) Insertion of the matched prefix 01011 into the 1st hash table



(b) Insertion of the prefix 11101 into the 2nd hash table

Removal of the duplicate of 01011 from the 2nd hash table



(c) The prefix 11111 does not replace the less popular prefix 10010 in the 3rd hash table

Figure 4.7: Light Traffic Hitters Detection. Example.

4.3.3 Light Traffic Hitters Detection

One of the biggest impediments to implement an efficient FIB caching architecture was the lack of an effective mechanism to quickly and accurately identify the unpopular routes for cache replacement. Due to extremely high rates of the network traffic, the flow monitoring by the control plane is associated with high bandwidth and computational costs.

In this work, we introduce a *Light Traffic Hitters Detection* module to find the least popular prefixes inside the data plane, without real-time scanning through the entire prefix list in the caches. Our approach is inspired by the algorithm described in [67] for finding n most popular flows ("Heavy-Hitters"). In their work, Sivaraman et al. propose $HashPipe$, that uses d disjoint hash tables $T_1, T_2, .., T_d$ in a pipeline for collecting and storing the heaviest flows at a line rate. Each table in $HashPipe$ is associated with an independent hash function $h_1, h_2, .., h_d$ which returns the ith slot in a corresponding table. When a packet arrives at the first table (Stage 1), $HashPipe$ calculates the hash key using h_1 function and the flow key

k. Suppose, the output of $h_1(k)$ is the slot i. If the ith slot is empty, then $HashPipe$ installs the flow key k and its corresponding counter value c into the first table. If the slot is occupied by a flow with the same key k, then its counter value is updated with c. If the slot is occupied by another flow with the key k' and it has a smaller counter value $c' < c$, k' is evicted from the first table and "carried" to the next hash table (Stage 2) along with its counter. The flow key k and the corresponding counter value is installed instead of k'. Otherwise, if the flow with the key k' is more popular than k, k flow is carried to Stage 2. The whole process is repeated until the last stage.

One side effect of $HashPipe$ is that duplicate entries may appear at different stages of the hash tables. For example, let $h_n(k)$ represent the hash value of key k in the nth hash table. Now, suppose $h_1(k_1) = h_1(k_2)$, where k_1 is a more popular flow than k_2 and k_1 occupies the ith slot in the first hash table. Assume at the second stage, the flow k_2 is installed in the jth slot. Later, the flow k_2 becomes more popular than k_1, and when the flow k_2 is carried through the first hash table, it displaces k_1 and its counter from the ith slot. Thus, the first and the second tables contain a duplicate entry with the flow k_2 (at ith and jth slots respectively).

Based on the similar idea in $HashPipe$, we designed a new pipeline of multiple stages to detect the least popular flows (i.e. prefixes) for quick cache replacement. In addition, we addressed hash entry duplicate issues, since (a) the duplicate entries occupy the slots that can be used for other flows; (b) duplicates complicate the hash entry removal upon cache eviction requests.

To collect the least popular prefixes, LTHD uses d disjoint hash tables $T_1, T_2, .., T_d$ in the data plane (see Figure 4.7). Each of the tables is assigned with a specific hash key $h_1, h_2, .., h_d$. On a cache hit, the matched prefix p and its counter value c is pipelined through the hash tables. Consequently, LTHD selects the most popular prefix among p and prefixes at in tables $T_1, T_2, .., T_d$ and replaces that prefix with p, unless p is not the most popular one. Next, we describe the mechanism of *Light Traffic Hitters* step-by-step.

We divide the operation of LTHD into d stages, where each stage corresponds to a hash table in the pipeline. Suppose, a prefix lookup in a cache resulted into a prefix p_1. In this case, PFCA updates p_1's counter c_1 and passes the tuple (p_1, c_1) to LTHD.

At the first stage, LTHD calculates the hash value $i = h_1(p_1)$. If the ith slot of T_1 is empty, LTHD installs (p_1, c_1) into T_1 and terminates. Otherwise, if it is occupied by another prefix (p_2, c_2), c_1 is compared against c_2. If $c_1 \geq c_2$, (p_1, c_1) is carried to the next stage of the pipeline. Otherwise, (p_2, c_2) is evicted from T_1 and replaced by (p_1, c_1). (p_2, c_2) is then carried to the next stage instead of (p_1, c_1). The rest of LTHD's operation is exact as at the first stage, i.e., the hash value $j = h_2(p_2)$ for the prefix p_2 and c_2 is compared against the counter c_3 of the prefix p_3 residing at the slot j in the table T_2. If $c_2 \geq c_3$, (p_2, c_2) is carried to the next stage; otherwise, (p_3, c_3) is replaced with (p_2, c_2) and carried to the next stage. To avoid duplicate

entries in the pipeline, in case the initially carried tuple (p_1, c_1) is installed at a certain stage n, LTHD checks the tables of the later stages $m <= d$ until the first duplicate of p_1 is found. Note that there is no need to carry the key that was evicted at an earlier stage n and installed at a later stage m, $d \geq m > n$, because when the key was installed in the nth table, the possible duplicates in any table $T_{n'}$, $d \geq n' \geq m > n$, should have been removed.

The pseudo-code of the first stage of LTHD is shown on Algorithm 7 (Appendix B). Figure 4.7 illustrates an example 3-stage LTHD. At the first stage, the prefix p_1=01011 with the hash $h_1(p_1)$=4 and counter c_1=2 is compared against the prefix p_2=11101 with counter c_2=3, since p_2 occupies the 4th slot in the first hash table. Since c_1¡c_2, the prefix p_1 replaces the prefix p_2. At the next stage, the duplicate of p_1 from the 2th slot is removed; in addition, the prefix p_2, carried from the first stage, replaces the more popular prefix p_3=11111. At the last stage, p_3 does not replace the prefix 10010 since its counter c_3 is greater by one. Thus, the prefix p_3 is evicted from the LTHD hash tables.

The pipelining in *Light Traffic Hitters Detection* happens in PFCA on each packet hit in Level-1 cache (TCAM) or Level-2 cache (SRAM). Note that PFCA uses different hash pipelines for TCAM and SRAM to separate victim prefixes for those forwarding tables.

4.4 P4 prototype

We built a partial prototype of PFCA in $P4_{16}$ data plane programming language [56] with respect to the Portable Switch Architecture (PSA) [58]. The design of PSA is fully compatible with the PFCA architecture. First, it allows the programmability of the ingress and egress pipelines. Next, a PSA switch can allocate necessary registers to implement *Light Traffic Hitters Detection* module. The forwarding tables in the ingress pipeline can be enriched with counters or meters per each entry, which enables heavy hitter detection with PFCA[1]. Alternatively, P4 registers can be used for detecting the heavy flows. Upon such detection, the ingress pipeline can generate a notification (via *packet digest* construction of PSA architecture) and send it to the control plane port. Finally, several PSA targets support parallel matching against multiple tables in the ingress pipeline [7, 73], as we design in PFCA with SRAM and DRAM forwarding tables.

In our P4 code, first, we defined the packet headers and the parser that decouples the Ethernet frame and IP headers from a packet. Second, we defined the match-action tables for both caches and the FIB on DRAM. Next, we defined the registers necessary for storing prefix counters and prefix hashes of *Light Traffic Hitters*

[1] According to the current specification of PSA, the values of the counters can not be read from the data plane. Implementing PFCA with PSA will be facilitated if such constraint is eliminated.

Detection module.

We show the snippets of the ingress pipeline code in Listing C.1 of Appendix C. Initially, we let ingress traffic pass through Level-1 cache. In case of a table hit, a packet's metadata field *miss_bit* is set to zero, so the control flow passes the packet's header to a the *Light Traffic Hitters Detection* module. The module is programmed in P4 using hashing functions available in the virtual P4 switch and the standard control flow commands provided by P4. In case of a Level-1 cache miss, *miss_bit* is set to one and the packet is matched against Level-2 cache and the slow FIB. A Level-2 cache hit is treated similarly to a Level-1 cache hit. We show the snippets of our P4 code for the Light Traffic Hitters Detection module in Listing C.2 of Appendix C.

To program the prototype of the Route Manager in the control plane, we used API generated by $P4_{16}$ compiler based on our P4 code. As the works on P4 PSA compiler are still in progress, we used a different switch model to run the emulation. We tested the partial PFCA implementation in bmv2 [57] virtual environment for P4-programmed switches. In our current P4 prototype, PFCA is able to forward packets and collect the victim routes into the hash tables of the *Light Traffic Hitters Detection* module. As the Route Manager detects heavy flows in Level-2 cache or the slow FIB, it randomly chooses victim entries and performs prefix migrations. In addition, the Route Manager handles the BGP update requests and applies them to the forwarding tables of the data plane.

4.5 Complexity

In this section, we analyze the complexity of the algorithms proposed in PFCA. Adding a new prefix with the length w into the binary tree at the Route Manager requires $2 * w$ memory accesses due to prefix extensions at each level of a tree between the root node and the new node. Hence, the running time of a BGP update handling is equal to $O(2 * w)$, when the new prefix is more specific than existing ones. In the meantime, in a worst-case scenario, a BGP update for a less specific prefix p may require a full post-order traversal of the tree rooted at a node corresponding to p. Such a traversal runs in $O(2 * n - 1)$, where n is the number of entries in the routing table with the extended prefixes. As we show in Section 4.6, the BGP updates in fact rarely affect popular entries in TCAM and thus the slight increase in the BGP update handling delay will not degrade the forwarding correctness of the cache entries.

The complexity of the *Light Traffic Hitters Detection* (LTHD) module is critical for the performance of a switch since it needs to track the least popular flows at high rates of the network traffic. Pipelining a matched prefix through LTHD requires $2 * d - 1$ memory accesses at most, where d is the number of stages set up

by the operator of a switch (in our evaluation, we use only 4 stages). The additional $d-1$ memory accesses stem from the necessity of removing a possible duplicate key from the tables $T_{m+1}, ..., T_d$ when installing a new key at a stage m (see Section 4.4).

4.6 Evaluation of PFCA

4.6.1 Experiment setup

For the experiment, we used the following realistic datasets: (1) A routing table with 599,298 entries from the RouteViews project [5]; (2) A one-hour traffic trace with 45,730 BGP updates from the RouteViews project; (3) A one-hour anonymized traffic trace collected by the CAIDA [1] with more than 3.5 billions of packets. All datasets were collected at 03/17/2016. By combining these three datasets we simulated a one-hour operation of an L3 switch/router with PFCA. For a full simulation, we coded PFCA in C, including the control plane with BGP updates handler, the Route Manager and the ingress pipeline with the *Light Traffic Hitters Detection* module. We ran the experiment on an Intel Xeon Processor E5-2603 v3 1.60GHz machine. The correctness of BGP update handling for an extended FIB table in PFCA was verified with VeriTable [32]. We evaluated our FIB caching architecture with the following metrics:

1. Cache-miss ratio per 100K packets for Level-1 and Level-2 caches.
2. The number of cache installations/evictions in Level-1 and Level-2 caches.
3. The number of BGP updates applied to Level-1 and Level-2 caches.

We did not measure cache-miss latencies in this experiment since the lookup latency depends on the switch target. However, the delay for a cache-missed packet can be estimated approximately as an LPM search time on TCAM ($\approx$4*ns* [78]).

4.6.2 Tuning the FIB caching architecture

Several parameters can affect the performance of PFCA. First, the number of stages in the *Light Traffic Hitters Detection* module can be changed, along with the size of each hash table, similarly to $HashPipe$ proposed in [67]. Decreasing the size of the hash tables and increasing the number of stages may increase the accuracy of the less popular routes selection. However, the tuning process is also constrained by the switch's

memory limit. Second, the popularity threshold for migrating a prefix to a faster memory may be changed in Level-2 (SRAM) cache and the FIB in DRAM. The threshold affects the frequency of installations into Level-1 (TCAM) and Level-2 caches, as well as the frequency of cache victim evictions and replacements when a faster memory is full. A low value of the threshold typically results in more frequent installations and victim evictions. Furthermore, we observed that frequent installations into a cache lead to an increasing number of cache misses. On the other side, the high value of the threshold deters installations of heavy flow prefixes into a faster memory, which might increase the number of cache-misses as well.

The optimal tuning of PFCA also depends on the memory size of TCAM/SRAM chips and the patterns of the traffic on the switch. While the theoretical approach to finding the optimal tuning parameters will constitute our future work, in this work we demonstrate the simulation results based on the following parameters:

1. For the *Light Traffic Hitters Detection* module, we set the number of stages to be 4, and the size of each hash table to be 10. Since each entry in the module contains a 4-byte prefix and a counter, the memory overhead for *Light Traffic Hitters Detection* is calculated as (4+4)$*$4$*$10=320 bytes. Because we use separate cache victim tables for Level-1 and Level-2 caches, the overall overhead for the switch memory is 640 bytes.
2. When the caches are not full, we set low thresholds in Level-2 cache and the slow FIB, 15 and 1, respectively, to quickly initialize them.
3. If Level-1 cache is full, the threshold in Level-2 cache is set to 300 matches per minute.
4. If Level-2 cache is full, the threshold in the slow DRAM FIB is set to 100 matches per minute.

In the simulation, we assume the limits of TCAM and SRAM memory sizes to be 20K and 40K entries, respectively. To verify the effectiveness of the chosen parameters, we ran a second experiment with a different traffic trace and obtained similar results.

4.6.3 Results

We started our experiment with the empty Level-1 and Level-2 caches. As it can be seen from Figure 4.11, both caches are filled quickly because of the low thresholds we set for cache initialization. During the experiment, PFCA restricts Level-1 and Level-2 caches to 20K and 40K maximum entries respectively.

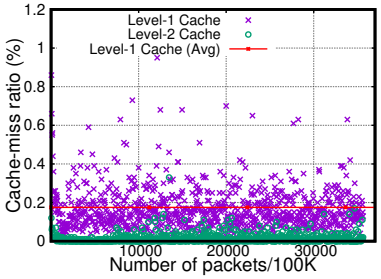


Figure 4.8: Cache-miss ratio with popularity-based victim selection.

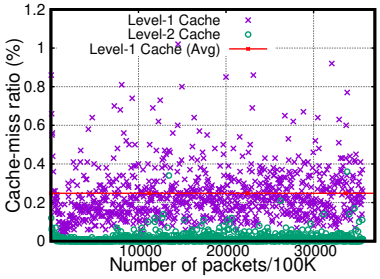


Figure 4.9: Cache-miss ratio with random victim selection

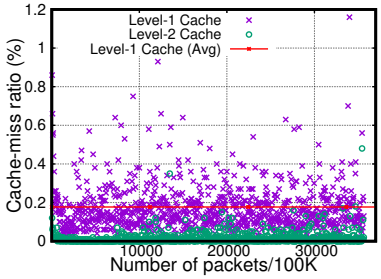


Figure 4.10: Cache-miss ratio with heap-based victim selection.

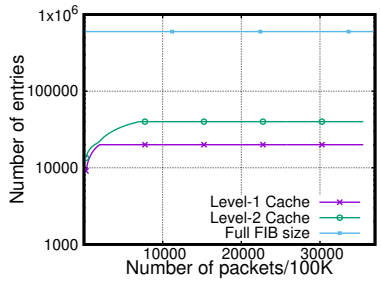


Figure 4.11: Size of Level-1, Level-2 caches and the full FIB.

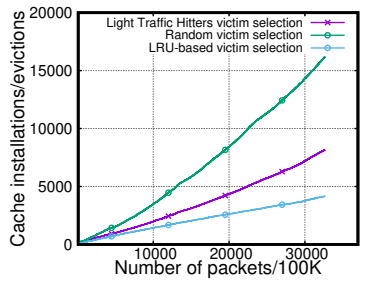


Figure 4.12: Installations and evictions in the full Level-1 cache.

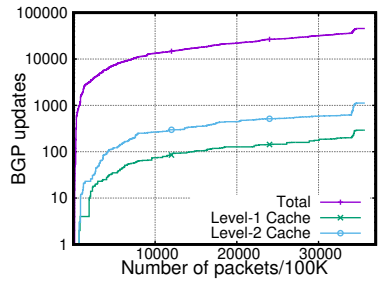


Figure 4.13: Number of BGP updates in the Level-1, Level-2 caches and the full FIB.

Cache-miss ratio

Cache-miss ratio is one of the most important characteristics to measure the performance of a FIB caching algorithm or framework. Figure 4.8 shows the cache-miss ratio with popularity-based victim selection for every 100K packets passing through the ingress pipeline of the switch. The least popular prefixes were collected by the *Light Hitter Detection* module. At the beginning of the experiment, the miss ratios for both cache tables were relatively high, since we did not initialize them up with empirically popular entries. However, the number of misses decreases quickly as the popular prefixes are filled into Level-1 and Level-2 caches. On average, the miss ratio for Level-1 cache is slightly less than 0.175%, and the miss ratio for Level-2 cache rarely exceeds 0.1% and is 0.015% on average. In general, the larger sizes of TCAM and SRAM caches, the lower cache-miss ratios for both caches.

Figure 4.9 shows the cache-miss ratio for Level-1 and Level-2 caches for a similar FIB architecture with random cache victim selection. The average miss ratio for Level-1 cache is 0.248% and the highest miss ratio after cache initialization is over 1%, which is much higher than that in the popularity-based victim

selection. For Level-2 cache, the average miss ratio is very close to the one for popularity-based victim selection ($\approx$0.015%). Such results for Level-2 cache can be explained by the observation that it consists of prefixes whose popularity are more uniformly distributed. However, note that the random eviction results are versatile and uncontrollable, thus it may lead to a large number of installations/evictions if a popular prefix was selected. On the contrary, our popularity-based victim selection approach will always choose relatively unpopular prefixes, and thus the results are more reliable and predictable.

In addition to evaluating random cache victim selection strategy, we simulated PFCA with the LRU-based cache victim selection policy, i.e., an algorithm that stores prefixes in a heap and selects the last recently matched prefix as a cache victim. Obviously, such an algorithm cannot be implemented in a switch due to its complexity. However, our goal was to compare the results of LRU-based victim selection with our *Light Traffic Hitters Detection* module. After running the simulation with the same data traces, we obtained the average cache-miss ratio for Level-1 cache slightly worse than cache-victim selection based on light traffic hitters (0.177% vs 0.175%). Similarly, for Level-2 cache, LRU-based victim selection resulted in more cache-misses than when using *Light Traffic Hitters Detection* module (0.017% vs 0.015%). Figure 4.10 shows the cache miss ratio of LRU-based victim selection for every 100K packets during the experiment. Overall, *Light Traffic Hitters Detection* achieves the smallest cache-miss ratio while using a line-rate algorithm.

Installations/evictions in a full cache

Recall that the installations are triggered when a prefix in a slower memory becomes popular, i.e., its counter reaches a threshold value. If the threshold is reached for a prefix in Level-2 cache, the prefix will be removed from it and installed into Level-1 cache; if the threshold is reached for a prefix in the FIB on DRAM, the prefix is installed into Level-2 cache and its $Cache$ flag is set. Note that if a cache is full, each installation requires eviction of another prefix from the cache.

Figure 4.12 illustrates the numbers of cache installations and evictions in a full Level-1 for (a) popularity-based, (b) random and (c) LRU-based victim selection implementations. As we can observe in the graph, the popularity-based approach yields nearly half of the number of cache installations and evictions, when compared to the random victim selection approach. As expected, the LRU-based victim selection approach, that chooses the least matched prefix as a cache victim, achieves the smallest value for installations and evictions. However, as we show in 4.6.3, *Light Traffic Hitters Detection* module achieves the minimal cash-miss ratio among those three approaches. On average, PFCA with the popularity-based approach installed only 8 entries in Level-1 cache and 20 entries in Level-2 cache for every million of data packets.

BGP updates impact

Figure 4.13 illustrates that 45,600 BGP updates triggered only 292 writes for Level-1 cache in TCAM and 1,120 writes for Level-2 cache in SRAM during a one-hour simulation. The results demonstrate that most of the BGP updates are not linked with popular routes in the caches, which indicates that FIB caching utilizes expensive routing memories more cost-effectively than existing practice. Intuitively, we can explain these results by the fact that most of the BGP instability comes from misconfigurations and failures, which is peculiar to the least popular destinations [22, 61].

An important advantage of PFCA is that it can minimize the number of expensive writing operations on TCAM since each write in TCAM triggers resorting of the entries on the chip. Meanwhile, the number of installations of popular routes is minimal once prefixes from heavy flows are installed into TCAM and prefixes from active flows are installed into SRAM (see Figure 4.12). Note that the number of installations into a table is equal to the number of evictions in case if the memory on a chip is full.

In general, the results show the efficiency of the proposed architecture. PFCA can achieve the minimal number of misses with only 3.5% of the global FIB installed on TCAM and 13.2% of the global FIB installed on SRAM, significantly reducing hardware and energy costs for these memory units.

4.7 PFCA. Discussion

One of the properties of PFCA is that its data plane operates with solely non-overlapping prefixes. Thus, for each destination IP address, there can be only a single match in the FIBs of PFCA, rather than multiple matches with different prefix lengths. This property cancels the necessity of conducting expensive Longest Prefix Matching (LPM) lookups. While generating a non-overlapping set of prefixes increases the size of a FIB and the churn caused by BGP updates, PFCA cancels these drawbacks by selecting a small subset of popular prefixes for forwarding more than 99% of traffic; as we show in Section 4.6, this subset consists of 5% of entries of a full FIB rarely affected by BGP updates. With the elimination of LPM, prefixes on TCAM no longer need to be sorted by their prefix lengths which significantly reduces the costs of entry insertions and deletions. Moreover, it is possible to leverage PFCA for software switches that do not use expensive TCAM memory and apply forwarding hash tables on a cheaper memory instead.

4.8 PFCA. Conclusion

In this chapter, we introduce a Programmable FIB Caching Architecture (PFCA) with two levels of FIB cache in TCAM and SRAM respectively, and the slow FIB in DRAM memory. A great advantage of this architecture is that operators can flexibly configure their TCAM and SRAM for efficient memory utilization. In addition, we proposed the *Light Traffic Hitters Detection* module that performs pipeline-based cache victim selection for cache replacement. We designed PFCA based on Portable Switch Architecture (PSA) and partially implemented its prototype using the P4 programming language. We demonstrated the effectiveness and efficiency of our architecture in terms of cache misses, route installations, evictions, and BGP updates handling. We employed two-hour traffic traces to evaluate and verify the PFCA's performance. According to the evaluation, PFCA achieves 99.825% cache-hit ratio for the Level-1 cache with 3.34% entries of the full FIB. Moreover, only 0.015% of the data packets were forwarded by the FIB on the slow DRAM memory, thanks to the Level-2 cache with 6.68% entries of the full FIB. Finally, 45,600 BGP updates included in the data trace resulted in merely 152 FIB changes in the Level-1 cache, showing that PFCA minimizes the BGP churn in the TCAM memory.

Chapter 5

CFCA: Combined FIB Caching and Aggregation

5.1 Introduction

The evaluation of the resulting FIB table in PFCA's Level-1 cache showed that it is aggregatable by at least 30% by an algorithm like FIFA-S [48] of FAQS. There are two sources of the mergeable entries:

1. The original global FIB contains a large amount of adjacent or overlapping prefixes that share the same next-hop.
2. To avoid *cache hiding* problem, when a more specific prefix hides in a secondary FIB table (which leads to incorrect LPM matching at the cache), most of FIB caching techniques [25, 30, 41, 47], including PFCA, perform conversion of a FIB into an equivalent FIB with fragmented non-overlapping prefixes.

A large number of adjacent prefixes in PFCA occupies additional space and increases the cache-miss ratio and cache churn, causing potential lookup latencies. This space can be freed with an FIB aggregation procedure (see example in Table 5.1). Yet, integrating FIB caching and FIB aggregation remains an uninvestigated problem [9]. There are two considerations that should be taken into account when designing a FIB aggregation in conjecture with FIB caching:

1. The FIB aggregation algorithm should not produce overlapping routes, that will lead to cache hiding

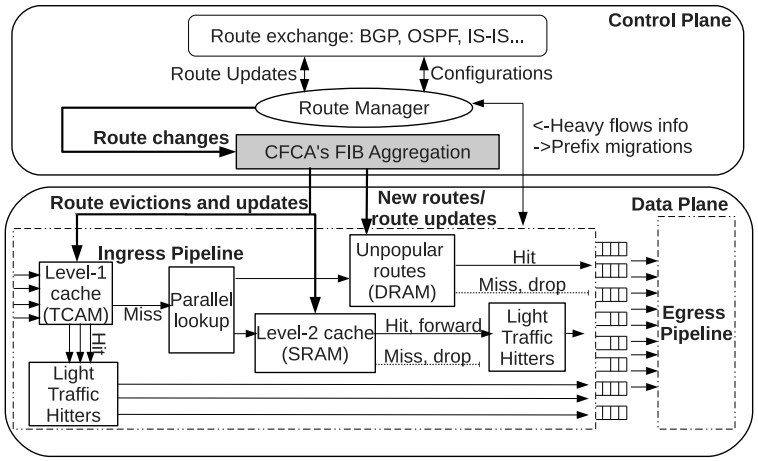


Figure 5.1: The architecture of CFCA

Label	Prefix	Next-hop
A	129.10.124.0/24	1
B	129.10.124.0/27	1
C	129.10.124.64/26	1
D	129.10.124.192/26	2

(a) Original FIB

Label	Prefix	Next hop
A	129.10.124.0/24	1
D	129.10.124.192/26	2

(b) Aggregated FIB

Table 5.1: Example of FIB aggregation

and incorrect forwarding behavior.

2. The FIB aggregation should incrementally handle BGP updates without significant increase of BGP churn in TCAM.

In this chapter, we present Combined FIB Caching and Aggregation - CFCA - that integrated caching and aggregation while addressing the above-mentioned considerations.

5.2 Design of CFCA

As shown in Figure 4.3, CFCA is built on top of Programmable FIB Caching Architecture (PFCA) [30]. The control plane of CFCA is responsible for collecting network routes and managing the tables in the data plane, namely Level-1 (L1) cache in TCAM, Level-2 (L2) cache in SRAM, and the rest of the FIB in DRAM. To avoid *cache hiding* (see Section 4.1 for details), the *Route Manager* (RM) extends the FIB

routes received from the network into a set of non-overlapping prefixes (see Figure 4.4). The main departure of CFCA from PFCA lies with the **addition of the FIB aggregation module** that compresses the extended prefix table of the RM into a smaller set of non-overlapping prefixes. The prefixes are aggregated via a binary prefix tree where prefixes and their next-hops are represented as nodes and the sibling nodes with the same next-hop merge into the parent node (see Figure 5.2). Important, the FIB aggregation module can incrementally handle BGP updates and push those updates into FIB tables with no risk of *cache hiding*. The FIB entries are provided with traffic counters to help detect frequently-accessed prefixes. A traffic counter is incremented for each match, and when a threshold is reached, the entry migrates to a faster cache table. In addition, the router keeps track of less popular routes (also known as light traffic hitters) in the cache for future cache replacement. The rest of the section discusses details of the RM, the FIB aggregation module, the data plane workflow, and the light traffic hitters detection algorithm.

5.2.1 Control plane workflow

The *Route Manager* (RM) first processes route updates from neighbor routers, and passes these updates to the FIB aggregation module. The RM then performs prefix migration by moving popular prefixes into fast memory and unpopular prefixes to slow memory. The FIB aggregation module serves as a filter between the RM and the data plane. It keeps the routing table in a compressed state by performing incremental FIB aggregation without introducing new overlapping routes nor changing the forwarding behavior. In the following sections, we discuss how to achieve this task at each stage of computation.

Initial FIB installation

At the initialization stage, CFCA's RM receives the current prefix table's snapshot - a Routing Information Base (RIB) - and extends it to get a set of non-overlapping prefixes, in order to avoid *cache hiding* due to the rule dependencies (see Section 4.1). Prefix extension (see Figure 4.4 as an example) is done by adding the prefixes to the binary prefix tree and generating additional nodes in order to get a full binary prefix tree. While some of the previous work used PATRICIA trie for cache hiding avoidance [47], for our first attempt of combined FIB caching and aggregation, we chose the binary tree data structure with pro-active installation of extended prefixes to prevent the overlapping prefixes with different next-hop values. With such an approach we aim to minimize frequency of node generation and removal in a tree after BGP updates.

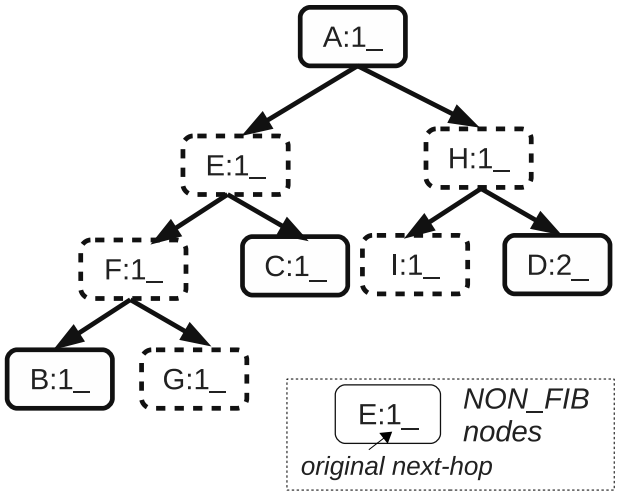
A binary prefix tree in CFCA has the following properties:

1. The root node *root* of the tree corresponds to the prefix 0/0 and is assigned with a default next-hop value.
2. Each node *n* in the tree may have zero or two children.
3. The left (right) child of a node *n.l* (*n.r*) corresponds to a more specific prefix with an appended *0*(*1*) bit.
4. The set of non-overlapping prefixes is composed by all the leaf nodes of the tree.

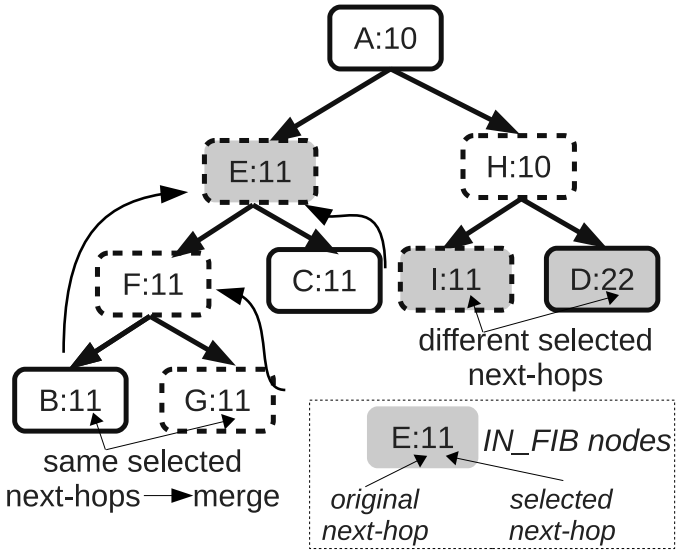
Each node *n* in CFCA's full binary tree has the following parameters:

1. The current table of a node, *n.t*. Possible values are *NONE*, *Level-1* (L1) or *Level-2* (L2) cache, or *DRAM*.
2. The prefix corresponding to that node, *n.p*.
3. *REAL*/*FAKE* flag *n.rf*, that indicates if a node was originated from the RIB (*REAL* node) or was generated as a result of prefix extension (*FAKE* node).
4. The *original* next-hop *n.o*, taken from the RIB. The *FAKE* nodes inherit the original next-hops of their *REAL* parent nodes.
5. The *selected* next-hop *n.s*, set by CFCA's FIB aggregation algorithm.
6. FIB status *n.f*: *IN_FIB* or *NON_FIB* which indicates if the corresponding prefix and its selected next-hop should be in installed into the data plane.

Although prefix extension does not change the forwarding behavior of a prefix table, it may significantly increase its size. **CFCA's aggregation algorithm** mitigates this negative effect by recursively "folding" adjacent prefixes that share the same next-hop into a larger prefix and installing into the FIB only one prefix per branch, thus preventing *cache hiding*. We show the pseudo-code of the initial FIB aggregation in Algorithm 8 of Appendix D. The compression is done within a single post-order traversal over binary prefix tree starting from the root node. Consider an example with entries from Table 5.1a. The initial binary prefix tree for these entries contains 5 leaf nodes (see Figure 5.2(a)). However, at the left branch, nodes *B* and *G* share the same next-hop from the RIB; thus, they merge into *F*. *F* selects its next-hop from *B* and *G*. Since that next-hop is equal to *C*'s next-hop, *F* and *C* similarly merge into *E*. At the right branch, leaves *I* and *D* have different next-hops from the RIB and do not merge; consequently, nodes *E* and *H* do not "fold" into *A* and the traversal terminates. As a result, instead of 5 prefixes, only 3 non-overlapping prefixes (from



(a) The prefix tree after prefix extension



(b) The prefix tree after FIB aggregation

Figure 5.2: FIB aggregation in CFCA

the nodes *E*, *I* and *D*) will be installed into the FIB. More specifically, at each node *n*, CFCA is running two operations: (1) Setting the selected next-hop value of *n*; (2) Setting the FIB status of the left and right children of *n*, namely *n.l* and *n.r*. Below is the detailed description of both operations.

1. **Setting the selected next-hop value *n.s* for the node *n*.** If *n* is a leaf node, then *n.s* is equal to the original next-hop value *n.o*. In case *n* is an internal node, its selected next-hop depends on the selected next-hops of *n.l* and *n.r*:

a. If *n.l.s*=*n.r.s*, then *n.s*=*n.l.s*;

b. Otherwise, *n.s*=0.

2. **Setting the FIB status of the left and right child nodes of n (*n.l.f* and *n.r.f*).** This operation is performed for the internal nodes of the prefix binary tree. In case *n.s* is not equal to 0, then both *n.l.f* and *n.r.f* are set to *NON_FIB*. Otherwise, if *n.s*$\neq$0, the FIB status of a child node is set to *IN_FIB* if its own selected next-hop value is not equal to 0. Initially, the entries corresponding to *IN_FIB* nodes are pushed into the DRAM memory of the data plane. While the traffic is passing through the data plane, those entries may migrate to L2 and L1 cache if they become popular.

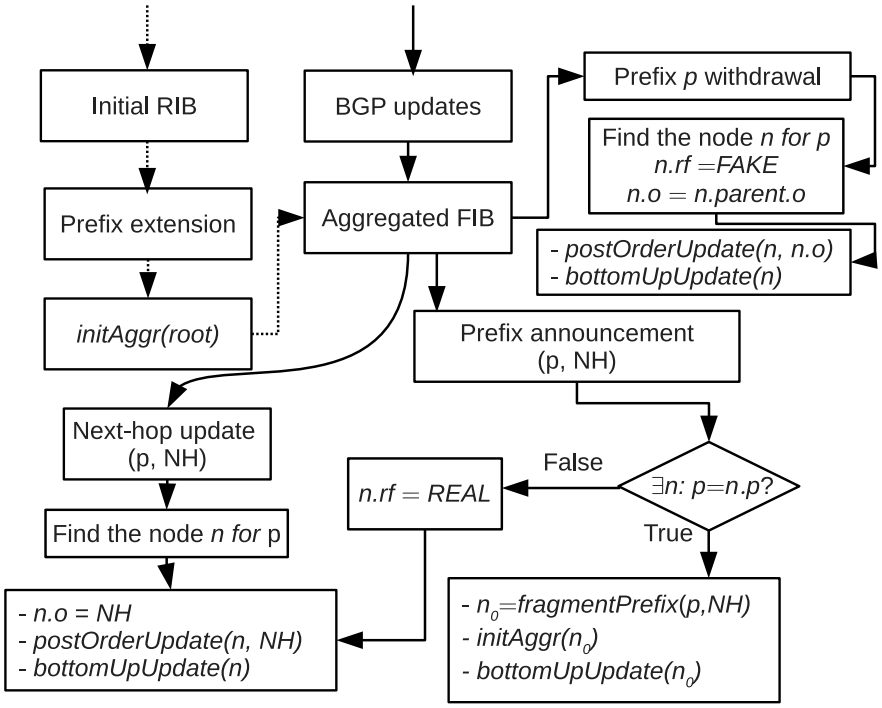
Intuitively, a non-zero selected next-hop value *n.s* of a node *n* means that all the leaf descendant nodes of *n* have the original next-hop value equal to *n.s*. Hence the FIB entries corresponding to those leaf prefixes can be compressed into a single entry. We call the nodes with a non-zero selected next-hop as *points of aggregation*. In the meantime, a zero value of *n.s* means that not all of the descendant leaf nodes of *n* have the same original next-hop value. If a left or a right child node of *n* is a point of aggregation, then it is assigned with *IN_FIB* status. At this stage, our algorithm guarantees that the prefix region $n.p$ is fully covered with a set of IN_FIB descendant nodes of *n*. Thus, *n* and its all ancestor nodes should be assigned with *NON_FIB* status.

CFCA's initial aggregation algorithm consists of a single post-order traversal of a binary tree, hence its complexity is $O(N)$, where N is the number of nodes in a binary tree. The number of nodes N is equal to $2 * N_p - 1$, where N_p is the total number of non-overlapping prefixes (i.e., leaf nodes). In the worst case, each original prefix p in a FIB can produce up to w non-overlapping prefixes, where w is p's prefix length. BGP update handling of CFCA has a similar complexity, as it leverages post-order traversals for keeping a FIB in a compressed state. We demonstrate it further in this subsection.

BGP update handling

The RM receives BGP updates (new entry announcements, next-hop updates, and entry withdrawals) from a BGP daemon running on the control plane. The RM applies these updates to the binary prefix tree by updating or adding a node to it. Next, CFCA re-aggregates the sub-tree rooted at the updated (or added) node, as well as the parents of that node (see Figure 5.3 for the workflow overview). However, in most cases, the update affects only a few nodes of a tree. Below we give a detailed description of how CFCA handles different types of BGP updates.

• ***Announcement of a new next-hop NH for an existing prefix p in a FIB***. A next-hop update of a prefix



p, p_0: Prefix n, n_0: Nodes

NH: Next-hop value $root$: The root node of the tree

Figure 5.3: BGP update handling workflow in CFCA

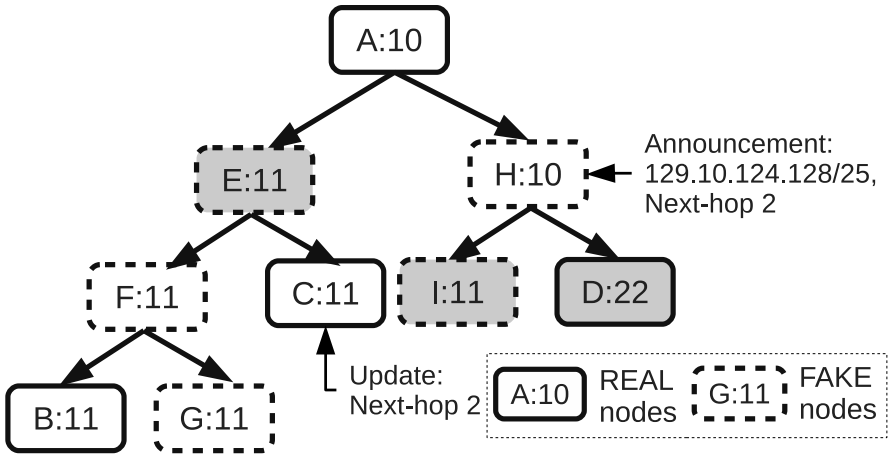
p may affect not just p itself, but also the prefixes generated after *prefix extension*. The resulting changes may require next-hop updates in the data plane, as well as de-aggregation or re-aggregation of prefixes overlapping with p. Thus, CFCA needs to perform partial traversals of a tree branch on which p resides. Next, we describe this process in details.

After the RM finds the node n corresponding to p in the prefix binary tree, it assigns the new value of the next-hop to $n.o$, the original next-hop of n. Next, it partially traverses the branch rooted at n in a post-order manner (see Algorithm 9 of Appendix D), avoiding the branches rooted at *REAL* nodes and visiting *FAKE* nodes only. Intuitively, the *REAL* descendants of n are not affected by the next-hop update of n and traversing them is useless. The *FAKE* nodes, on the contrary, are generated during prefix extension in CFCA. Since they inherit next-hop values from n, they are updated with every change to $n.o$.

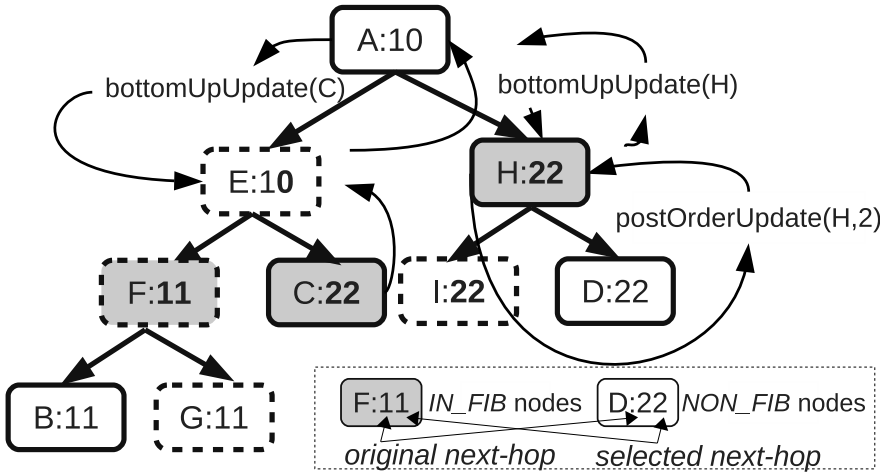
During the traversal, at each traversed node, CFCA:

1. Updates the node's original next-hop with *NH*.
2. Sets the selected next-hop of the node in the same manner as in the *Initial FIB Aggregation*. More specifically, for the leaf nodes, the selected next-hop is equal to their original next-hops. For the internal nodes, the selected next-hop is equal to 0 if their children nodes have different selected next-hop values. The procedure *setSelectedNextHop(n)* is shown in Algorithm 12 of Appendix D.
3. Sets the FIB status of the node's children (if they exist) and pushes the next-hop changes to the data plane. More specifically, if a node has a non-zero selected next-hop, its children should not be present in the FIB. Otherwise, the children nodes with a non-zero selected next-hops (i.e., points of aggregation) should stay or be installed into the data plane. Note that the RM tracks the current location of entries with the value of $n.r.t$ and thus is able to push the updates to the correct data plane destination (L1, L2 caches or DRAM). We give the pseudo-code of *setFIBstatus(n)* in Algorithm 13 of Appendix D.

Once the post-order traversal is done, the ancestors of the node n should be traversed in a bottom-up manner. There are two reasons why such procedure is necessary: first, the ancestors' selected next-hop value depends on their children's selected next-hops. Second, in CFCA, the FIB status of a node is defined by its parent. Since the changes in a tree propagate through selected next-hops, the bottom-up traversal can terminate if a node's selected next-hop does not change. Similarly to the post-order traversal, at each visited node n, CFCA calls both *setSelectedNextHop(n)* and *setFIBstatus(n)*. We show the pseudo-code of *bottomUpUpdate(n)* in Algorithm 10 of Appendix D. Note that *bottomUpUpdate(n)* is called only if the value of $n.s$ is changed after calling *setSelectedNexthop(n)*.



(a) Update for C and announcement of a prefix at H



(b) Nodes E, F, C, H, I, D have been affected

Figure 5.4: BGP updates handling by CFCA

Although a new route announcement and a route withdrawal require more operational steps by the RM, they mostly rely on the same routines as with a route update, i.e., *postOrderUpdate(n, NH)* and *bottomUpUpdate(n)*. We show it below in this section.

• ***Announcement of a new route (p, NH), where p is the prefix and NH is its next-hop value***. Two cases may happen. In the first case, a node *n* associated with *p* already exists in the binary prefix tree. CFCA handles such announcements similarly to the next-hop updates. The RM updates *n*'s original next-hop value, changes its type to *REAL* (if it was *FAKE* prior to the update) and runs *postOrderUpdate(n, NH)* and *bottomUpUpdate(n)* routines.

In the second case, the node *n* for *p* should be first added to the binary prefix tree. For that goal, the RM traverses the tree accordingly to the bits of *p*, to find n_0, the existing leaf ancestor for *n*. Next, the RM installs the node *n* while generating *FAKE* sibling nodes for each level of the tree between n_0 and *n*. Such generation is necessary to avoid *cache hiding*. These siblings nodes will inherit their original next-hop value from n_0 and their prefix values will belong to the prefix range of n_0 with exclusion of p. We call this procedure *prefix fragmentation*, since it fragments the prefix of n_0 into a set of non-overlapping prefixes with *p*. The psedo-code of prefix fragmentation is given in Algorithm 11 of Appendix D.

The newly added nodes need to be aggregated, thus CFCA calls the initial aggregation *aggrInit* starting from the node n_0. Finally, in case the selected next-hop of the node n_0 is changed, CFCA runs *bottomUpUpdate(n_0)* to re-aggregate the upper part of the branch, until the first unchanged ancestor.

• ***Withdrawal of a prefix p***. Suppose *n* is the node that represents *p* in the binary prefix tree. CFCA handles prefix withdrawals as follows:

1. The Route Manager sets the type of *n* to *FAKE*.
2. The node inherits its original next-hop value from the parent node. This value might be a default next-hop, inherited from the root node of the tree, or an arbitrary next-hop from a less specific prefix existing in the original RIB.
3. Similarly to the next-hop updates, the aggregation module runs *postOrderUpdate(n, NH)* and *bottomUpUpdate(n)* routines.

Intuitively, the prefix withdrawal operation in CFCA can be represented as a node's original next-hop update with that node's parent's original next-hop value. In the meantime, CFCA ensures the compacted state of the binary prefix tree, by detecting and removing sibling *FAKE* leaf nodes from the data structure.

In Figure 5.4, we illustrate BGP update handling by CFCA. We continue using the entries from Table 5.1a. In this example, the RM received two BGP updates: a next-hop update for the prefix *129.10.124.64/26* and an announcement of a new prefix *129.10.124.128/25*. For the first update, CFCA reaggregates the tree with bottom-up traversal from the node *C*, after which the nodes *E*, *F* and *C* change their FIB status. For the announcement of the *129.10.124.128/25*, the algorithm first changes the type of the corresponding node *H* from *FAKE* to *REAL*. Next, it performs the post-order and bottom-up traversals from the node *H*, after which the prefixes at the nodes *I* and *D* aggregate into *H*.

Popular routes installation

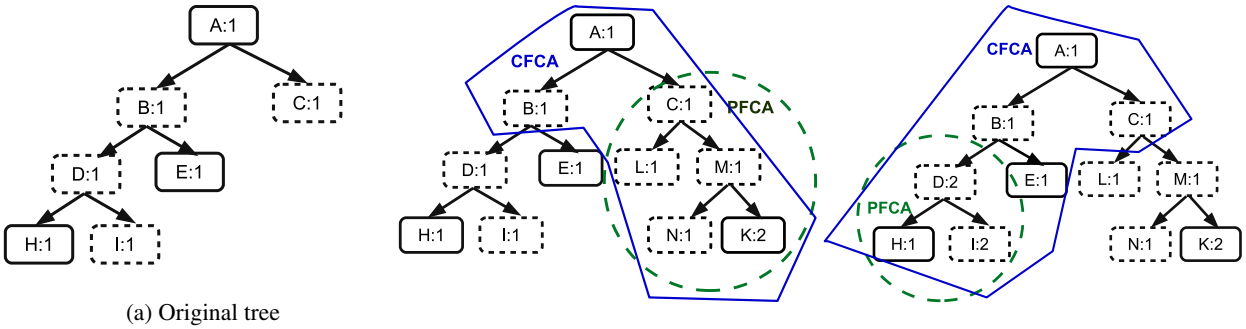
The RM ”migrates” the entries of a FIB to faster memory once a route in the L2 cache table or the DRAM reaches a certain amount of hits, for example, 100 hits per second. The threshold can be pre-configured by the control plane of a switch. If a faster memory is full, then the RM frees it by picking a victim cache entry with the help of *Light Traffic Hitters Detection* module of the data plane and moving that entry to slower memory.

5.2.2 Data plane workflow

Similarly to PFCA, the essential part of the CFCA's data plane is the *Light Traffic Hitters Detection* (LTHD) module, designed in order to collect the least popular routes of Level-1 (L1) and Level-2 (L2) caches for future cache replacement. LTHD can be built in the switches with the Portable Switch Architecture [58]. Its design is described in details in 4.3.3 of Chapter 4. The flow between the match-action tables of the data plane (L1, L2 caches and the DRAM FIB) is managed by the programmable ingress pipeline. When a packet arrives at a CFCA switch, its destination IP address is matched against the L1 cache FIB on TCAM. In the case of a match, the matching prefix's counter is increased by one and its value is passed through the LTHD module. The packet is then forwarded to the next-hop.

In the case of an L1 cache miss, the packet's IP header is passed to the L2 cache and the DRAM table in parallel. On an L2 cache match, CFCA increases the counter value of the matching prefix and checks if it reached a pre-configured threshold. If so, the prefix is being installed into the L1 cache. In case the L1 cache is full, CFCA randomly picks an unpopular route from the LTHD module's tables and replaces it with a new route from the L2 cache. Lastly, the unpopular prefix from the L1 cache migrates back to the L2 cache. If the threshold on the L2 cache for a matched prefix is not reached, CFCA passes the prefix and its counter value through the LTHD module of the L2 cache. In each case, a match in the L2 cache results in the forwarding of the packet to the next-hop, corresponding to the matched prefix.

Finally, if the match happens on DRAM, then, similarly to the L2 cache, CFCA checks if the matched prefix's counter value reached a pre-configured threshold and installs that prefix into the L2 cache if needed. As with the L1 cache, if the L2 cache is full, the cache victim entry is selected from the LTHD module and migrated into the FIB on DRAM.



(a) Original tree

(b) Node K added

(c) Node D updated

Figure 5.5: Comparison of node accesses between PFCA and CFCA (worst case scenario)

5.2.3 Complexity of CFCA's aggregation

As CFCA is built upon PFCA (see Section 4) with addition of the aggregation layer, it comes with additional costs for BGP update handling compared to pure FIB caching. Consider two cases:

1. Addition of a new prefix with the length w into the binary tree. In such a worst-case scenario, PFCA needs simply to generate $2 * w$ nodes to avoid cache hiding. In addition to generating the nodes and recursive bottom-up aggregation up to the first *REAL* ancestor node, CFCA will need to perform re-aggregation of the updated branch (total $O(2 * 32 - 1)$ node accesses in the worst case, when the bottom-up update propagates up to the root node).

2. Updating an existing node with length λ and n derived leaf nodes. While PFCA, similarly to the first case, does not perform any bottom-up operations, in the worst case, it will need to access $O(2 * n - 1)$ leaf nodes. CFCA will have to perform $O(\lambda * 2 - 1)$ additional node accesses to reaggregate the ancestors of the updated note.

Figure 5.5 illustrates number of nodes accessed by both algorithms in a specific example.

5.3 CFCA. Evaluation

We evaluate the performance of CFCA by comparing it to pure FIB caching (PFCA) and FIB aggregation algorithms with incremental BGP updates handling (FAQS and FIFA-S):

- PFCA is a FIB caching-only architecture for a programmable switch described in [30]. We use PFCA for comparison in terms of cache-miss ratio and BGP-related FIB changes.
- FAQS [46] is a FIB aggregation algorithm with minimal data plane churn and suboptimal aggregation ratio.
- FIFA-S [48] is an ORTC-based FIB aggregation algorithm with an optimal aggregation ratio.

To emulate the operation of a programmable switch with CFCA we designed its data plane prototype with P4 language. As works on the P4 compiler for PSA are in progress [69], our prototype was designed for the Simple Switch architecture [70], and thus had limited capabilities compared to the original design of CFCA. First, "Simple Switch" lacks the generation of packet digests in a data plane, i. e., messages that can be forwarded to a control plane, which is necessary for detecting the heavy hitters at the latter. Another obstacle for building the full prototype with P4's "Simple Switch" is that the packet counters attached to the entries in routing tables can not be accessed directly from the data plane.

While our future work will focus on designing the full P4 prototype of CFCA based on PSA, in this work, we evaluate the efficiency of CFCA's combined FIB caching and aggregation using a trace-driven simulator written in C.

5.3.1 Experimental setup

We use RouteViews [5] to build a routing table that contains 599,298 entries. Our experiment simulates router operations for about an hour by processing a mixed trace of 45,600 BGP updates of RouteViews and an anonymized traffic trace from CAIDA [1] with 3.5 billion packets. For performance evaluation, we measure the following metrics:

1. Cache-miss ratio per 100K packets for Level-1 (L1) and Level-2 (L2) caches.
2. The number of popular route installations/evictions in L1 and L2 caches.
3. The number of BGP updates applied to L1 and L2 caches compared to the total number of BGP updates.

We tune both CFCA and PFCA in the same way as follows:

1. We set the number of stages in the *Light Traffic Hitters Detection* (LTHD) module to four.

CFCA/ PFCA	L1 size ratio, %	L1 size	L2 size	L1 misses, %	L2 misses, %	L1 installations	L2 installations	L1 BGP updates	L1 BGP max burst
CFCA	0.83	5,000	10,000	1.133	0.153	109,070	623,994	88	2
	1.67	10,000	15,000	0.144	0.017	18,108	150,547	156	5
	2.50	15,000	20,000	0.058	0.004	17,277	37,842	285	6
PFCA	0.83	5,000	10,000	4.135	1.258	302,337	664,580	51	8
	1.67	10,000	15,000	0.873	0.211	72,400	118,847	133	12
	2.50	15,000	20,000	0.316	0.071	32,242	58,246	195	12

Table 5.2: CFCA vs FIB caching with PFCA. Evaluation summary.

2. Each stage has a hash table of size 10.
3. Initially, L1 and L2 caches are empty; however, they are quickly filled up as the initial threshold for route installations in DRAM and Level-2 cache are 1 and 15 matches, respectively.
4. Once the L1 and L2 caches are full, we set the thresholds to 100 matches per minute for DRAM and to 300 matches per minute for L2 cache.
5. We compare the performance of CFCA and PFCA for different sizes of L1 and L2 caches, namely 5K with 10K, 10K with 15K, and 15K with 20K.

We verified the correctness of BGP update handling by CFCA, PFCA, FAQS and FIFA-S with VeriTable [32] (see Section 3.4.1), a tool designed for fast verification of forwarding equivalence of multiple forwarding tables.

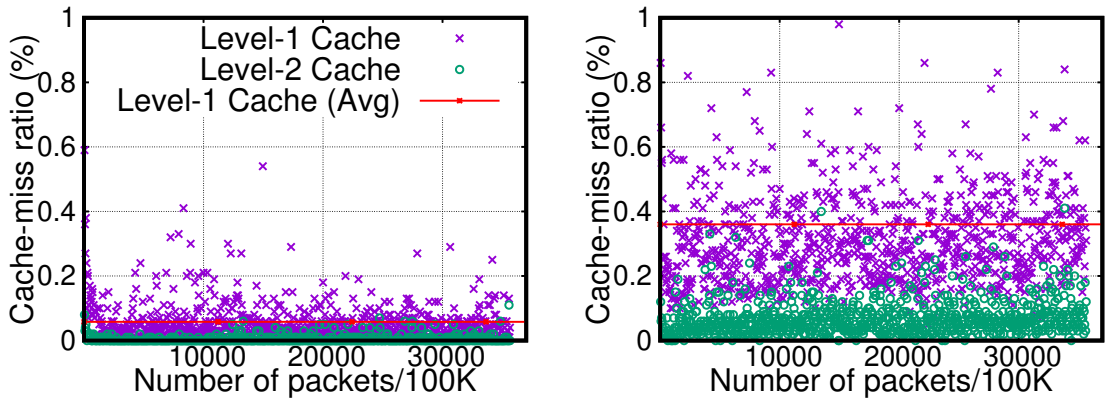
5.3.2 CFCA vs FIB Caching (PFCA)

The results that compare CFCA with PFCA are shown in Table 5.2. In the table,

1. *L1/L2 size ratio, %*: the ratio between the maximum number of entries in L1/L2 cache and the total number of entries;
2. *L1/L2 size*: the maximum number of entries in L1/L2 cache.
3. *L1/L2 misses, %*: the ratio between successful LPM lookups and the total number of LPM lookups in a table.

CFCA/FAQS/FIFA-S	Size compared to the original FIB, %	FIB updates	BGP max burst
CFCA	2.50	21,495	6
FAQS	28.04	36,386	43
FIFA-S	25.08	48,672	33

Table 5.3: CFCA L1 cache vs FAQS/FIFA-S



(a) CFCA

(b) PFCA

Figure 5.6: CFCA vs PFCA. Cache-miss ratio per 100K packets for 15,000 L1 and 20,000 L2 caches

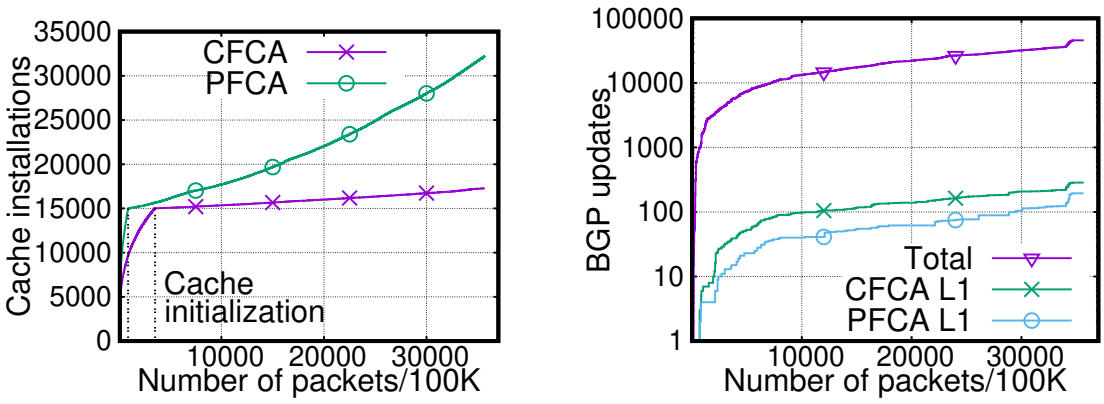
4. *L1/L2 installations*: the total number of popular route installations into L1/L2 caches.

5. *L1 BGP updates*: the total number of changes in TCAM caused by BGP protocol.

6. *L1 BGP max burst*: the maximum observed number of FIB changes after a single BGP route update during the experiment.

Cache-miss ratio

We first measure cache-miss ratios. Here, the larger values are, the more IP lookups are to be performed at slow memory causing packet forwarding delays and even losses. It is ideal to have the L1 cache miss ratio is minimized. In CFCA, the average cache-miss ratio is 1.13% when the L1 cache contains only 0.83% entries of the FIB. Also, if the L1 cache contains more prefixes (2.50% of the FIB), the cache-miss ratio drops drastically to 0.058%, which is more than five times less than PFCA's average cache-miss ratio (0.316%). Figure 5.6 shows this cache-miss ratio improvement through FIB aggregation (see the red line



(a) Popular route installations and evictions

(b) Number of BGP updates

Figure 5.7: L1 cache in CFCA and PFCA

for the average). The number of L1 cache-miss lookups in CFCA rarely exceeds 0.4%, while the number of cache-misses is negligible for L2. In contrast, the cache-miss ratio of PFCA reaches nearly 1% in the worst case.

Popular route installations and evictions into L1/L2 caches

In CFCA, the control plane aggregates FIB entries before pushing them into the data plane, and this aggregation dramatically decreases both L1 and L2 popular route installation and eviction rates. The low installation and eviction rates help save energy and improve performance as writing on TCAM is computationally expensive [34]. We show the L1 popular route installations for both PFCA and CFCA in Figure 5.7a. We set the sizes of the L1 and L2 caches to 15,000 and 20,000 entries, respectively. A cache entry installation does not require eviction until the cache is fully initialized (see the dashed lines in the figure). Note that both CFCA and PFCA apply the cold start strategy, i.e., L1 and L2 cache tables are initially empty and are filled up with traffic later. Once the cache tables become full, cache replacement is needed for a new popular route installation, and we use *Light Traffic Hitters Detection* to identify victims for eviction.

BGP updates impact

One drawback of FIB aggregation is the potential cache churn increase in TCAM due to the BGP updates applied to a compressed FIB. Table 5.2 shows that BGP-related cache updates in the L1 cache of CFCA are slightly increased compared to PFCA because PFCA does not aggregate the FIB. However, the percentage of TCAM updates in CFCA remains low. As an example, only 0.625% of all BGP updates result in TCAM

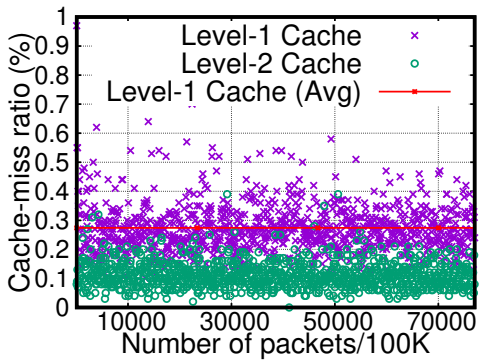


Figure 5.8: CFCA. Cache-miss ratio per 100K packets under a heavier load

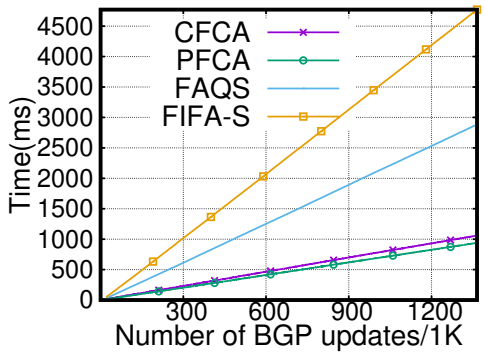


Figure 5.9: BGP update handling time

updates when the L1 cache size is 15,000 (see Figure 5.7b). In addition, we evaluate BGP maximum burst in L1, i.e., the maximum number of FIB changes caused by a single BGP update. Interestingly, CFCA shows lower BGP maximum burst in all the configurations as shown in Table 5.2. The FIB aggregation compresses prefixes generated by the prefix extension, and therefore a single next-hop update of a parent prefix affects fewer entries in TCAM than PFCA with more extended prefixes present in TCAM.

5.3.3 CFCA vs FIB Aggregation (FAQS/FIFA-S)

FIB aggregation minimizes its disadvantages when applied together with FIB caching. Table 5.3 shows that the total number FIB updates, which includes L1 popular route installations, evictions, and next-hop updates, in CFCA with 15,000 entries is far less than in FAQS, that is a FIB aggregation scheme with the minimal BGP-related churn. For ORTC-based algorithms with optimal aggregation ratio like FIFA-S, the total number of FIB updates is even larger. Moreover, CFCA does not produce large BGP bursts, which we attribute to the observation that BGP updates are mostly related to unpopular routes. The only strength of FIB aggregation over FIB caching is that all the routes stay in a cache. In FIB caching, on the other hand, the lookups that do not find a prefix match in a cache are forwarded to lower memory obviously increasing forwarding delay. CFCA mitigates this drawback as it reduces the cache-miss ratio to 0.058% for the L1 cache with only 2.50% entries in the FIB.

5.3.4 Evaluating a heavier trace

In the previous subsections, we showed the performance of CFCA under a traffic load with the mean transmission rate of about 1M packets per second. To test the behavior of CFCA under a heavier load, we

CFCA/PFCA	L1 cache size, %	L1 misses, %	L1 installations	L1 BGP max burst	Total BGP cost, s
CFCA	2.50	0.27	40,315	5	1.07
PFCA	2.50	0.81	162,737	18	0.93

Table 5.4: CFCA vs PFCA (larger trace)

collected a more recent (01/17/2019) 1-hour trace from CAIDA [1] with 7.7 billions of packets and a 1-hour BGP update trace with 1,364,963 routing updates. We also used a larger routing table with 725,344 entries and L1/L2 cache with 20,000 and 30,000 entries respectively.

Due to significantly increased packet frequency, the average cache-miss ratio for this trace is higher ($\approx$0.27%). However, CFCA keeps it within 1% (see Figure 5.8 and Table 5.4). For this trace, we also compared the BGP update handling time by CFCA vs PFCA, FAQS and FIFA-S (see Figure 5.9). CFCA's control plane takes only 0.78μs per a BGP update on average, slightly slower than PFCA (0.69μs per update), due to the additional FIB aggregation. In the meantime, the time cost of FIB aggregation algorithms is much higher (2.11μs and 3.50μs per update for FAQS and FIFA-S respectively).

5.4 CFCA. Conclusion

In this chapter, we present CFCA, Combined FIB Caching and Aggregation, which uses two FIB compressing techniques to offload the expensive TCAM memory used for FIB from unpopular routes. Evaluation of CFCA on large traffic trace and BGP update traces shows remarkably low cache-miss ratios and total number of FIB updates compared to the existing FIB caching solutions. In particular, CFCA achieves up to 99.94% cache-hits and stabilizes the cache by more than 46% with 2.5% of the entire FIB in the cache. Finally, compared to FIB aggregation algorithms, CFCA avoids large bursts of TCAM updates, reducing costly TCAM updates by 40% on average.

Chapter 6

Related Work

TCAM overflow problem and possible approaches for solving it were analyzed in in [80]. While IP protocol re-designing cannot be applied solely at a local ISP level and requires consistency between all of the Autonomous Systems (ASes), both FIB aggregation and FIB caching are local solutions and can significantly prolong the lifetime of a routing device. Specifically, a network operator can run FIB caching and aggregation techniques only within its AS without changing the traffic behavior within or outside that AS.

6.1 FIB aggregation

A number of FIB aggregation algorithms have been proposed. We highlight a few of them here. *SMALTA* algorithm [72] uses the binary tree data structure and bases on *ORTC* algorithm [23], which can achieve one-time optimal aggregation. *SMALTA* takes *ORTC* as the initial FIB aggregation algorithm and processes updates without the optimization of a subtree, rooted at the updated node. Eventually, *SMALTA* requires full re-aggregation of the FIB table upon reaching a FIB size threshold. It results in computational spikes and high time costs. In [65], authors study and employ the locality of FIB updates to build Locality-aware FIB Aggregation (LFA) algorithm. In LFA, reaggregation for an updated prefix region is delayed until it is stabilized. However, such an approach requires timers attached to nodes which may significantly complicate its operation in the real routers. Bienkowski et al. [10] present a formal study on the trade-off between FIB aggregation and update bursts. In addition, the paper presents the algorithm *HIMS* that attaches time-dependent counters to each node as well. However, the paper provides no information on the performance of the algorithm when processing real network routing data. In [36], authors propose MMS, the Memory Management System designed to prolong the lifetime of legacy routers in an ISP. MMS uses parallelization

of ORTC and can aggregate routing tables locally or on an AS-level. Moreover, MMS may change the forwarding behavior of routers in order to gain additional compression. Such an approach, in combination with aggregation algorithms, may result in the minimal FIB size. However, it cannot be deployed locally due to the potential forwarding loops.

Some FIB compression work uses smart data structures to minimize storage size of FIB [59]. In [60], authors present a tunable aggregation algorithm with compressed prefix trees. By changing the deepness of the compression, network operators can manage the trade-off between the aggregation ratio and BGP update overhead. Similarly, Yang et al. in [77] present two algorithms, *EAP-slow* and *EAP-fast* and compare it with *ORTC*. Unlike FAQS, EAP algorithms use leaf-pushed binary tree data structure. EAP achieves faster aggregation by traversing the leaf-pushed trie only once. In [49], authors propose an aggregation algorithm for OpenFlow flow tables, including non-prefix rules and prefix-based rules. For prefix-based rules, wildcard aggregation technique is used. FIB aggregation scheme, that applies multiple selectable next hops, is proposed by Li et al. [43]. Abraham et al. [4] create a virtual network system to implement and study FIB aggregation. It is a reusable framework to test the performance of FIB aggregation algorithms in a realistic environment.

Aggregation algorithms such as *Level-1* and *Level-2* [79] compress FIB quickly but bear costly update handling operations. In 2013, Liu et al. developed *FIFA* algorithms [48], which improves *ORTC* algorithm by applying PATRICIA tree (PT) with incremental FIB aggregation features. The basic of *FIFA* algorithms, *FIFA-S*, reduces FIB size by at least 70% for an IPv4 routing table. The faster update handling is achieved by applying formally proved properties, which allow to considerably reduce the number of FIB entries to be accessed. The main weakness of *FIFA-S* is the high number of changes in a FIB per routing announcement and related processing delays. Overall, the average number of FIB changes can be twice as more than the total number of FIB updates. Challenging FIB update churn problem,

Our work, *FAQS* algorithm [46], makes a balance of aggregation time, ratio and memory consumption. It sacrifices very little aggregation ratio compared with the optimal solution but speeds up the aggregation more than twice with much less memory consumption.

6.2 FIB caching

The idea of caching entries in a forwarding table goes back to the early stages of the Internet. In [35], Raj Jain *et al.* showed the ”source locality” of packets and proposed caching FIB entries for active flows and prefetching entries for the corresponding reverse flows. Route caching was studied for Layer-4 switching

as well [75]. Kim *et al.* give an overview of route caching in [41]. The paper studies traffic patterns in a large ISP network and measures the performance of route caching when applied to that ISP. Similarly, Gadkari *et al.* study FIB caching in [25]. In addition, the paper proposes a FIB caching solution that reaches 99.5% cache-hit ratio with 10K cache size, given the size of a FIB less than 500K entries. Both papers show that FIB caching achieves higher hit ratio under LRU policies. However, LRU cache victim selection may become a bottleneck when handling real Internet traffic. Alternatively, Sarrar *et al.* in [64] propose Traffic-aware Flow Offloading (TFO) strategy that outperforms LRU for a cache with less than 10,000 entries. The heavy-hitter selection process is performed at the control plane and may create a bottleneck when processing ISP traffic. In [63], authors apply rule caching for lossy compression of packet classifiers and show that 3% of the original classifiers are sufficient for handling 99% of the traffic. The paper does not elaborate on a chosen real-time cache victim selection strategy. Liu *et al.* in [47] use OpenFlow's variable idle timeouts to identify unpopular routes in the fast memory. The proposed strategy achieves 99.95% cache-hit ratio with 5.28% of the original FIB rules. Similarly to our work, the authors solve the cache hiding problem by prefix extensions. Katta *et al.* in *Cacheflow* [38] build dependency graphs and identify independent groups of rules to avoid cache hiding. Such an approach requires the control plane to perform graph computations each time a flow becomes popular and needs to be installed in the cache. Similarly, Bienkowski *et al.* in [11] propose an online *Tree Caching* algorithm, that requires no prefix extensions of the FIB. Evaluation of *Cacheflow* showed 97% cache-hit rate with 10%-sized TCAM; the authors of *Tree Caching* do not provide experimental results but instead give a theoretical analysis of its algorithm. Both algorithms are suited for a data center Software-Defined Networks with medium-sized FIBs. Our work, PFCA [30] leverages the programmable data plane in order to build a FIB caching architecture with two levels of cache. The data plane workflow and the *Light Traffic Hitters Detection* module of PFCA allow to install and remove entries from TCAM cache at the line rate.

In [62], Rottenstreich et al. propose lossy packet classifier compressors for optimal rule caching, that can be used for identifying the most popular flows in a FIB and offloading TCAM from idle flows. The *approximate classification* achieves the optimal cache-hit ratio given the size of the cache. However, it allows false classifications for cache-misses which can lead to the incorrect forwarding behavior of a router. The *cached classification* achieves the full forwarding correctness by redirecting some of the lookups to slow memory. Both algorithms are based on dynamic programming with non-linear time complexity, which may impede the deployment of the optimal rule caching in backbone networks with constantly changing traffic patterns and thousands of BGP updates per second.

IHARC [17] speeds up software IP lookup by utilizing CPU cache for route caching. It merges prefixes in the cache by selecting index bits that point to different cache sets with maximum mergeability. However, IHARC's architecture has several shortcomings. First, IHARC leverages a 3-level NART (Network Address

Routing Table) data structure, with prefixes, flattened to the lengths 8, 24 and 32. This significantly increases the size of a routing table and consequent update churn. Second, the index selection routine is an expensive operation invoked at each routing update. The updates require IHARC to entirely invalidate its caches for maintaining forwarding correctness, increasing cache-miss ratios. In [27], the authors improve IHARC with selective cache invalidations and reduced recomputations of index bits.

Chapter 7

Conclusion

7.1 Discussion

The global routing table growth poses risks of Internet disruptions and forces network operators to upgrade their TCAM chips. The widespread deployment of IPv6 protocol will increase the FIB size even further. In this work, we take two different approaches to mitigating the FIB overflow problem. First, we studied FIB aggregation and proposed a new FIB aggregation algorithm, FIB Aggregation with Quick Selections (*FAQS*), that significantly reduced the time costs of BGP update handling and the BGP churn on TCAM, compared to the existing algorithm. We evaluated *FAQS* on a 6-year BGP traces and show it achieves near-optimal FIB compression.

Second, we studied FIB caching. For the first time, we leveraged the programmable data plane concept to propose a new Programmable FIB Caching Architecture (*PFCA*) with cache-miss lookups handled entirely in the data plane. Furthermore, we proposed a new cache victim collection algorithm, that collects the light-flow prefixes into the data plane's registers for future cache replacements. We built the prototype of *PFCA*'s victim selection mechanism in P4 data plane language. We simulated *PFCA* on real network traces and showed that it achieved 99.8% cache-hit ratio and minimized the BGP churn compared to the FIB aggregation solutions.

When analyzing the resulting FIB table in the caches of *PFCA*, we noticed that algorithms like *FAQS* can compress it by 30%. The freed space in the cache may maximize the cache-hit ratio. However, the problem of integrating FIB caching and aggregation had not been studied yet. In addition, the existing FIB aggregation algorithms, including FAQS, could not be directly applied along with FIB caching, since they would

have produced overlapping prefixes which would lead to *cache hiding* and incorrect forwarding behavior of the data plane. To this end, we designed and evaluated the Combined FIB caching and Aggregation (CFCA) architecture with two main components:

- An FIB aggregation layer, that (a) compresses the forwarding table before pushing it into the data plane; (b) incrementally handles BGP updates to keep the FIB compressed; (c) prevents the presence of overlapping routes in the data plane.
- An FIB caching layer, that offloads TCAM from the unpopular routes.

CFCA avoids producing overlapping routes by using the binary tree data structure and preventing prefixes from the same branch to be installed into the cache and the main FIB. In CFCA, cache-hit ratio is maximized up to 99.94% with only 2.50% entries of the FIB, while the churn in TCAM is reduced by more than 40% compared to low-churn FIB aggregation techniques. We verified our experiments with VeriTable, a tool we designed for fast verification of forwarding equivalence and presented in this work. The promising results motivate us to implement the full prototype of CFCA programmable switch hardware built with respect to Portable Switch Architecture.

7.2 Limitations

While we built a limited prototype of the FIB caching architecture presented in this document, we were not able to build the complete prototype for programmable switch emulator due to several reasons. First, the Portable Switch Architecture's compiler works are in progress, thus we could not leverage its properties such as sending packet digests to the control plane, which is necessary to make the control plane aware of "hot" entries in the routing table. In the meantime, table entries, as well as counters and meters attached to those entries are not accessible from within the data plane. The simple addition of such functionality will make PFCA and CFCA able to move the entries between different tables without invoking the control plane.

7.3 Future work

With respect to the limitations listed in the above subsection, we plan to work with academia and industry community in order to build the full prototype of CFCA in a programmable switch emulator and later on programmable switch hardware. In the meantime, we plan to optimize our control plane code. In particular,

optimizations can be applied to the data structure we use for the FIB aggregation, as well as the reduction of memory accesses during the post-order aggregation traversals of CFCA.

Bibliography

[1] Center for Applied Internet Data Analysis. http://www.caida.org/home/, 2017.

[2] P4 Language Consortium. https://p4.org/, 2019.

[3] The 768k or Another Internet Doomsday? Here's how to deal with the TCAM overflow at the 768k boundary. https://www.noction.com/blog/768k-day-512k-tcam, 2019.

[4] Jerald Paul Abraham, Yaoqing Liu, Lan Wang, and Beichuan Zhang. A flexible quagga-based virtual network with FIB aggregation. *IEEE Network*, 28(5):47–53, 2014.

[5] Advanced Network Technology Center and University of Oregon. The RouteViews project. http://www.routeviews.org/, 2017.

[6] Ishfaq Ahmad and Sanjay Ranka. *Handbook of Energy-Aware and Green Computing-Two Volume Set*. Chapman and Hall/CRC, 2016.

[7] Barefoot. Tofino P4-programmable switch. https://www.barefootnetworks.com/products/brief-tofino/, 2019.

[8] BGP Routing Table Analysis Report. Active BGP entries (FIB). http://bgp.potaroo.net/, 2019.

[9] Marcin Bienkowski, Jan Marcinkowski, Maciej Pacut, Stefan Schmid, and Aleksandra Spyra. Online tree caching. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 329–338. ACM, 2017.

[10] Marcin Bienkowski, Nadi Sarrar, Stefan Schmid, and Steve Uhlig. Competitive FIB aggregation without update churn. In *Proceedings of the 34th International Conference on Distributed Computing Systems*, pages 607–616. IEEE, 2014.

[11] Marcin Bienkowski, Nadi Sarrar, Stefan Schmid, and Steve Uhlig. Online aggregation of the forwarding information base: Accounting for locality and churn. *IEEE/ACM Transactions on Networking (TON)*, 26(1):591–604, 2018.

[12] Roberto Bifulco and Anton Matsiuk. Towards scalable sdn switches: Enabling faster flow table entries installation. *ACM SIGCOMM Computer Communication Review*, 45(4):343–344, 2015.

[13] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[14] Brian Field. Approaches for IPv4 as a Service. In *2015 Spring Technical Forum Proceedings*. Comcast, 2015.

[15] H.J. Chao and B. Liu. *High Performance Switches and Routers*, pages 5–9. Wiley - IEEE. Wiley, 2007.

[16] Ray CC Cheung, Manish K Jaiswal, and Zahid Ullah. Z-TCAM: An SRAM-based architecture for TCAM. *IEEE Transactions on very large scale Integration (VLSI) systems, Digital Object Identifier*, 10, 2015.

[17] Tzi-Cker Chiueh and Prashant Pradhan. Cache memory design for network processors. In *Proceedings of the 6th International Symposium on High-Performance Computer Architecture*, pages 409–418. IEEE, 2000.

[18] Cisco Systems. How to Choose the Best Router Switching Path for Your Network . {https://www.cisco.com/c/en/us/support/docs/ip/express-forwarding-cef/13706-20.html}, 2005.

[19] Cisco Systems. Troubleshooting Prefix Inconsistencies with Cisco Express Forwarding. http://www.cisco.com/c/en/us/support/docs/ip/express-forwarding-cef/14540-cefincon.html, 2005.

[20] Cisco Systems. CAT 6500 and 7600 Series Routers and Switches TCAM Allocation Adjustment Procedures. https://www.cisco.com/c/en/us/support/docs/switches/catalyst-6500-series-switches/117712-problemsolution-cat6500-00.html, 2014.

[21] Cisco Systems. Configuring CEF Consistency Checkers. http://www.cisco.com/c/en/us/td/docs/ios-xml/ios/ipswitch_cef/configuration/12-4/isw-cef-12-4-book/isw-cef-checkers.html, 2016.

[22] Shivani Deshpande, Marina Thottan, Tin Kam Ho, and Biplab Sikdar. An online mechanism for bgp instability detection and analysis. *IEEE Transactions on Computers*, 58(11):1470–1484, 2009.

[23] Richard Draves, Christopher King, Srinivasan Venkatachary, and Brian D. Zill. Constructing Optimal IP Routing Tables. In *Proceedings of the 1999 IEEE Conference on Computer Communications*, 1999.

[24] Tobias Enderle and Uwe Bauknecht. Modeling dynamic traffic demand behavior in telecommunication networks. In *Photonic Networks; 19th ITG-Symposium*, pages 1–8. VDE, 2018.

[25] Kaustubh Gadkari, M Lawrence Weikum, Dan Massey, and Christos Papadopoulos. Pragmatic router fib caching. *Computer Communications*, 84:52–62, 2016.

[26] Geoff Huston. BGP in 2016. `https://blog.apnic.net/2017/01/27/bgp-in-2016/`, 2017.

[27] Kartik Gopalan and Tzi-cker Chiueh. Improving route lookup performance using network processor cache. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, pages 22–22. IEEE, 2002.

[28] Greg Hankins. FIB Scaling: A Switch/Router Vendor Perspective. `https://archive.nanog.org/meetings/nanog39/presentations/fib-hankins.pdf`, 2007.

[29] G. Grigoryan and Y. Liu. Toward a programmable fib caching architecture. In *2017 IEEE 25th International Conference on Network Protocols (ICNP)*, pages 1–2, 2017.

[30] Garegin Grigoryan and Yaoqing Liu. Pfca: a programmable fib caching architecture. In *Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems*, pages 97–103. ACM, 2018.

[31] Garegin Grigoryan, Yaoqing Liu, and Minseok Kwon. Boosting FIB Caching Performance with Aggregation. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, page 221–232. ACM, 2020.

[32] Garegin Grigoryan, Yaoqing Liu, Michael Leczinsky, and Jun Li. Veritable: Fast equivalence verification of multiple large forwarding tables. In *Proceedings of the 2018 IEEE Conference on Computer Communications*, pages 621–629. IEEE, 2018.

[33] John Hawkinson and Tony Bates. Guidelines for creation, selection, and registration of an Autonomous System (AS). Technical report, MCI Inc., 1996.

[34] Peng He, Wenyuan Zhang, Hongtao Guan, Kavé Salamatian, and Gaogang Xie. Partial order theory for fast tcam updates. *IEEE/ACM Transactions on Networking (TON)*, 26(1):217–230, 2018.

[35] Raj Jain and Shawn Routhier. Packet trains–measurements and a new model for computer network traffic. *IEEE journal on selected areas in Communications*, 4(6):986–995, 1986.

[36] Elliott Karpilovsky, Matthew Caesar, Jennifer Rexford, Aman Shaikh, and Jacobus Van Der Merwe. Practical network-wide compression of ip routing tables. *IEEE Transactions on Network and Service Management*, 9(4):446–458, 2012.

[37] Naga Katta, Omid Alipourfard, Jennifer Rexford, and David Walker. Infinite cacheflow in software-defined networks. In *Proceedings of the 3rd workshop on Hot topics in software defined networking*, pages 175–180. ACM, 2014.

[38] Naga Katta, Omid Alipourfard, Jennifer Rexford, and David Walker. Cacheflow: Dependency-aware rule-caching for software-defined networks. In *Proceedings of the 2016 Symposium on SDN Research*, page 6. ACM, 2016.

[39] Peyman Kazemian, George Varghese, and Nick McKeown. Header Space Analysis: Static Checking for Networks. In *Proceedings of the 2012 Symposium on Networked Systems Design and Implementation*, volume 12, pages 113–126. USENIX, 2012.

[40] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P Godfrey. Veriflow: Verifying network-wide invariants in real time. *ACM SIGCOMM Computer Communication Review*, 42(4):467–472, 2012.

[41] Changhoon Kim, Matthew Caesar, Alexandre Gerber, and Jennifer Rexford. Revisiting Route Caching: The World Should Be Flat. In *PAM*, volume 9, pages 3–12. Springer, 2009.

[42] Diego Kreutz, Fernando MV Ramos, Paulo Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, 2015.

[43] Qing Li, Mingwei Xu, Dan Wang, Jun Li, Yong Jiang, and Jiahai Yang. Nexthop-selectable FIB aggregation: An instant approach for internet routing scalability. *Computer Communications*, 67:11–22, 2015.

[44] Wenjun Li, Dagang Li, Xinwei Liu, Ting Huang, Xianfeng Li, Wenxia Le, and Hui Li. A power-saving pre-classifier for tcam-based ip lookup. *Computer Networks*, 164:106898, 2019.

[45] Alex X Liu, Chad R Meiners, and Eric Torng. Packet classification using binary content addressable memory. *Biological Cybernetics*, 24(3):1295–1307, 2016.

[46] Yaoqing Liu and Garegin Grigoryan. Toward incremental fib aggregation with quick selections (faqs). In *Proceedings of the 17th International Symposium on Network Computing and Applications (NCA)*, pages 1–8. IEEE, 2018.

[47] Yaoqing Liu, Vince Lehman, and Lan Wang. Efficient fib caching using minimal non-overlapping prefixes. *Computer Networks*, 83:85–99, 2015.

[48] Yaoqing Liu, Beichuan Zhang, and Lan Wang. Fifa: Fast incremental fib aggregation. In *Proceedings of the 2013 IEEE Conference on Computer Communications*, pages 1–9. IEEE, 2013.

[49] Shouxi Luo, Hongfang Yu, and Lemin Li. Practical flow table aggregation in sdn. *Computer Networks*, 92:72–88, 2015.

[50] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[51] Danny McPherson, Shane Amante, and Lixia Zhang. The Intra-domain BGP Scaling Problem. *RIPE 58, Amsterdam*, 2009.

[52] Chad R Meiners, Alex X Liu, Eric Torng, and Jignesh Patel. Split: Optimizing space, power, and throughput for tcam-based classification. In *Proceedings of the 2011 ACM/IEEE Seventh Symposium on Architectures for Networking and Communications Systems*, pages 200–210. IEEE Computer Society, 2011.

[53] Xiaoqiao Meng, Zhiguo Xu, Beichuan Zhang, Geoff Huston, Songwu Lu, and Lixia Zhang. IPv4 address allocation and the BGP routing table evolution. *ACM SIGCOMM Computer Communication Review*, 35(1):71–80, 2005.

[54] Mehrdad Moradi, Feng Qian, Qiang Xu, Z Morley Mao, Darrell Bethea, and Michael K Reiter. Caesar: High-speed and memory-efficient forwarding engine for future internet architecture. In *Proceedings of the 2015 ACM/IEEE Symposium on Computer Communications*, pages 171–182. IEEE, 2015.

[55] Donald R Morrison. PATRICIA - practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM (JACM)*, 15(4):514–534, 1968.

[56] P4 Language Consortium. $P4_{16}$ Language Specification. `https://p4lang.github.io/p4-spec/docs/P4-16-v1.0.0-spec.pdf`, 2017.

[57] P4 Language Consortium. Behavioral Model Repository. `https://github.com/p4lang/behavioral-model`, 2018.

[58] P4 Language Consortium. Portable Switch Architecture (PSA). `https://p4.org/p4-spec/docs/PSA.html`, 2019.

[59] Gábor Rétvári, Zoltán Csernátony, Attila Körösi, János Tapolcai, András Császár, Gábor Enyedi, and Gergely Pongrácz. Compressing ip forwarding tables for fun and profit. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, pages 1–6. ACM, 2012.

[60] Gábor Rétvári, János Tapolcai, Attila Kőrösi, András Majdán, and Zalán Heszberger. Compressing ip forwarding tables: Towards entropy bounds and beyond. *ACM SIGCOMM Computer Communication Review*, 43(4):111–122, 2013.

[61] Jennifer Rexford, Jia Wang, Zhen Xiao, and Yin Zhang. Bgp routing stability of popular destinations. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, pages 197–202. ACM, 2002.

[62] Ori Rottenstreich and János Tapolcai. Optimal rule caching and lossy compression for longest prefix matching. *IEEE/ACM Transactions on Networking*, 25(2):864–878, 2016.

[63] Ori Rottenstreich and János Tapolcai. Optimal rule caching and lossy compression for longest prefix matching. *IEEE/ACM Transactions on Networking (TON)*, 25(2):864–878, 2017.

[64] Nadi Sarrar, Steve Uhlig, Anja Feldmann, Rob Sherwood, and Xin Huang. Leveraging zipf's law for traffic offloading. *ACM SIGCOMM Computer Communication Review*, 42(1):16–22, 2012.

[65] Nadi Sarrar, Robert Wuttke, Stefan Schmid, Marcin Bienkowski, and Steve Uhlig. Leveraging locality for fib aggregation. In *Proceedings of the 2014 Global Communications Conference*, pages 1930–1935. IEEE, 2014.

[66] Damien Saucez, Luigi Iannone, Olivier Bonaventure, and Dino Farinacci. Designing a deployable internet: The locator/identifier separation protocol. *IEEE Internet Computing*, 16(6):14–21, 2012.

[67] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, Shan Muthukrishnan, and Jennifer Rexford. Heavy-hitter detection entirely in the data plane. In *Proceedings of the 2017 Symposium on SDN Research*, pages 164–176. ACM, 2017.

[68] Ahsan Tariq, Sana Jawad, and Zartash Afzal Uzmi. TaCo: Semantic Equivalence of IP Prefix Tables. In *Proceedings of the 20th International Conference on Computer Communications and Networks*, pages 1–6. IEEE, 2011.

[69] The P4 language consortium. Compiler for PSA. `https://github.com/p4lang/behavioral-model/issues/873`, 2020.

[70] The P4 language consortium. The BMv2 simple switch target. `https://github.com/p4lang/behavioral-model/blob/master/docs/simple_switch.md`, 2020.

[71] Inayat Ullah, Zahid Ullah, and Jeong-A Lee. Ee-tcam: An energy-efficient sram-based tcam on fpga. *Electronics*, 7(9):186, 2018.

[72] Zartash Afzal Uzmi, Markus Nebel, Ahsan Tariq, Sana Jawad, Ruichuan Chen, Aman Shaikh, Jia Wang, and Paul Francis. SMALTA: practical and near-optimal FIB aggregation. In *Proceedings of the 2011 Conference on emerging Networking EXperiments and Technologies*, 2011.

[73] V. Gurevich. Programmable Data Plane at Terabit Speeds. {https://conferences.sigcomm.org/sigcomm/2018/files/slides/p4/P4Barefoot.pdf}, 2017.

[74] Vladimir Gurevich. Programmable Data Plane at Terabit Speeds.

[75] Jun Xu, Mukesh Singhal, and Joanne Degroat. A novel cache architecture to support layer-four packet classification at memory access speeds. In *Proceedings of the 2000 IEEE Conference on Computer Communications*, volume 3, pages 1445–1454. IEEE, 2000.

[76] Tong Yang, Jinyang Li, Chenxingyu Zhao, Gaogang Xie, and Xiaoming Li. Mathematical analysis on forwarding information base compression. *CCF Transactions on Networking*, 1(1-4):16–27, 2019.

[77] Tong Yang, Bo Yuan, Shenjiang Zhang, Ting Zhang, Ruian Duan, Yi Wang, and Bin Liu. Approaching optimal compression with fast update for large scale routing tables. In *Proceedings of the 20th International Workshop on Quality of Service*, IWQoS '12, pages 32:1–32:9, Piscataway, NJ, USA, 2012. IEEE Press.

[78] Fang Yu, Randy H Katz, and Tirunellai V Lakshman. Gigabit rate packet pattern-matching using tcam. In *Proceedings of the 12th IEEE International Conference on Network Protocols, 2004*, pages 174–183. IEEE, 2004.

[79] X. Zhao, Y. Liu, L. Wang, and B. Zhang. On the Aggregatability of Router Forwarding Tables. In *Proceedings of the 2010 IEEE Conference on Computer Communications*, 2010.

[80] Xiaoliang Zhao, Dante J Pacella, and Jason Schiller. Routing scalability: an operator's view. *IEEE Journal on Selected Areas in communications*, 28(8):1262–1270, 2010.

Appendices

Appendix A

Algorithms of FAQS

Algorithm 1 shows the pseudo-code of the initial FIB aggregation via post-order traversal in *FAQS*. Algorithm 2 shows the process of selection of a new next hop for a PT node. Next, algorithm 3 shows the pseudo-code of *FAQS*'s decision making on installing a prefix into the data plane. Finally, Algorithm 5 show the pseudo-code of BGP update handling by *FAQS*.

Algorithm 1 Static FIB Aggregation

```
1:  procedure StaticAggregation(node)
2:      p ← node.parent
3:      l ← node.left
4:      r ← node.right
5:      if T(node) ≠ REAL then
6:          O(node) ← O(p)
7:      end if
8:      if l ≠ NULL then
9:          StaticAggregation(l)
10:     end if
11:     if r ≠ NULL then
12:         StaticAggregation(r)
13:     end if
14:     setSelectedNexthop(node)
15:     setChildFIBstatus(node)
16: end procedure
```

Algorithm 2 Assignment of Selected Next Hop

```
1: procedure SetSelectedNexthop(node)
2:     l ← node.l
3:     r ← node.r
4:     if l ≠ NULL ∧ r ≠ NULL∧
         len(l) − len(node) = 1∧
         len(r) − len(node) = 1∧
         O(node) ≠ S(r) then
5:         S(node) ← S(l)
6:     else
7:         S(node) ← O(node)
8:     end if
9: end procedure
```

Algorithm 3 Determine FIB status for Children Nodes

```
1: procedure SetChildFIBstatus(node)
2:     l ← node.l
3:     r ← node.r
4:     if l ≠ NULL then
5:         if S(node) ≠ S(l) then
6:             F(l) ← IN_FIB
7:         else
8:             F(l) ← NON_FIB
9:         end if
10:    end if
11:    if r ≠ NULL then
12:        if S(node) ≠ S(r) then
13:            F(r) ← IN_FIB
14:        else
15:            F(r) ← NON_FIB
16:        end if
17:    end if
18: end procedure
```

Algorithm 4 Incremental FIB Update Handling

```
1: procedure UpdateNode(prefix, nexthop)
2:     node ← find(prefix)
3:     if node = NULL then
4:         node ← initialize(prefix, nexthop)
5:         T(node) ← REAL
6:         p ← node.parent
7:         if O(p) ≠ O(node) then
8:             UpdateSubtree(node)
9:             UpdateAncestors(node)
10:        end if
11:    else
12:        if T(node) ≠ REAL then
13:            T(node) ← REAL
14:        end if
15:        if O(node) ≠ nexthop then
16:            O(node) ← nexthop
17:            UpdateSubtree(node)
18:            UpdateAncestors(node)
19:        end if
20:    end if
21: end procedure
```

Algorithm 5 Update Subtree

```
1: procedure UpdateSubtree(node)
2:     l ← node.l
3:     r ← node.r
4:     if l ≠ NULL ∧ T(l) ≠ REAL then
5:         O(l) ← O(node)
6:         UpdateSubtree(l)
7:     end if
8:     if r ≠ NULL ∧ T(r) ≠ REAL then
9:         O(r) ← O(node)
10:        UpdateSubtree(r)
11:    end if
12:    setSelectedNexthop(node)
13:    setChildFIBstatus(node)
14: end procedure
```

Algorithm 6 Update Ancestors

```
1:  procedure UpdateAncestors(node)
2:      p ← node.parent
3:      while p ≠ NULL do
4:          oldSlctNexthop ← S(p)
5:          setSelectedNexthop(p)
6:          setChildFIBstatus(p)
7:          if oldSlctNexthop = S(p) then
8:              break
9:          end if
10:         p ← p.parent
11:     end while
12: end procedure
```

Appendix B

Algorithms of PFCA

On Algorithm 7 we present PFCA's cache victim selection procedure, described in Section 4.3.3.

Algorithm 7 Light Traffic Hitters Detection (Stage 1)

1: **procedure** $LTHD(p, c)$ ▷ $p \leftarrow$ prefix, $c \leftarrow$ its counter; $d \leftarrow$ number of stages; $h_1, h_2, .., h_d \leftarrow$ independent hash functions; $T_1, T_2, .., T_d \leftarrow$ hash tables.
2: $i = h_1(p)$
3: **if** ith slot of T_1 is empty **then**
4: Install (p, c) into T_1
5: Carry p to remove possible duplicates in $T_2..T_d$.
6: **goto** the next stage
7: **else if** ith slot in T_1 is occupied by (p, c_0) **then**
8: $c_0 = c$
9: **else if** ith slot in T_1 is occupied by $(p', c') \wedge c < c'$ **then**
10: Replace (p', c') by (p, c) in T_1
11: Carry (p', c') for installation in the next stages
12: Carry p to remove possible duplicates in $T_2..T_d$. **goto** the next stage
13: **else if** ith slot in T_1 is occupied by $(p', c') \wedge c \geq c'$ **then**
14: Carry (p, c) to the next stage.
15: **goto** the next stage
16: **end if**
17: **end procedure**

Appendix C

Listings for Light Traffic Hitters Detection Module

The following listings present PFCA's P4 code listings for Section 4.3.3.

Listing C.1: Ingress pipeline at PFCA

```
{if (hdr.ipv4.isValid()){
level1.apply();
if (meta.miss_bit == 1){
level2.apply();
if (meta.miss_bit == 1){
dram.apply();
if (counter > threshold){
/* Building a packet digest for
the control plane */
meta.packet_digest.prefix = prefix;
meta.packet_digest.memType = DRAM;
}

}
else {
if (counter > threshold){
meta.packet_digest.prefix = prefix;
meta.packet_digest.memType = L2;
}
/* Go to LTHD for Level-2 */
}
}
else
/* Go to LTHD for Level-1 */
...
}
```

Listing C.2: Light Traffic Hitters Detection at Level-1 cache

```
/* Defining registers and the hash function */
Register<bit<32>,bit<32>>(20) lthd_keys_1;
Register<bit<32>,bit<32>>(20) lthd_keys_2;
Register<bit<8>,bit<32>>(20) lthd_counters_1;
Register<bit<8>,bit<32>>(20) lthd_counters_2;
Hash<bit<32>>(PSA_HashAlgorithm_t.CRC32) h1;
Hash<bit<32>>(PSA_HashAlgorithm_t.IDENTITY) h2;
.......
/* LTHD module */
/* Stage 1 for a tuple (prefix, counter) */
hashed_prfx = h1.get_hash(prefix);
prefix_key_1 = lthd_keys_1.read(hashed_prfx);
prefix_counter_1 = lthd_counters_1.read(hashed_prfx);

/* Case 1: the slot is occupied by the same prefix */
if (prefix_key_1 == meta.matched_prefix){
/* Update the counter and leave the pipeline */
lthd_counters_1.write(hashed_prfx, counter);
carry = 0;
}
/* Case 2: the slot is empty */
else if (prefix_counter_1 == 0){
/* Install the key and the counter
and leave the pipeline */
lthd_keys_1.write(hashed_prfx, prefix);
lthd_counters_1.write(hashed_prfx, counter);
carry = 0;
}
/* Case 3: the slot is occupied by
a heavier flow */
else if (prefix_counter_1 > counter){
/* Replace the tuple (prefix_key_1, prefix_counter_1)
by (prefix, counter) and move to Stage 2*/
lthd_keys_1.write(hashed_prfx, prefix);
lthd_counters_1.write(hashed_prfx, counter);
carry = 1;
}
/* Case 4: the slot is occupied by
a lighter flow */
else {
/* Carry (prefix, counter) to the next stage */
prefix_key_1 = prefix;
prefix_counter_1 = counter;
carry = 1;
}
if (carry == 1){
/* Stage 2 for a tuple
(prefix_key_1, prefix_counter_1) */
....
}
```

Appendix D

Algorithms of CFCA

Below we present the algorithms described in Section 5.2. Algorithm 8 shows the pseudo-code of the initial aggregation in CFCA. Algorithm 9 and 10 shows the pseudo-code of post-order and bottom-up traversals in CFCA upon a BGP update. Algorithm 12 shows the pseudo-code of next hop selection process for a node in CFCA. Algorithm 11 shows the pseudo-code for prefix fragmentation in CFCA, when a more specific prefix than existing in the routing table is added via BGP. Finally, Algorithm 13 shows the pseudo-code of FIB status selection process in CFCA.

Algorithm 8 Static FIB Aggregation in PFCA

procedure $aggrInit(n)$ ▷ $n \leftarrow$ node, $n.o \leftarrow$ its original next-hop; $n.s \leftarrow$ its selected next-hop; $n.f \leftarrow$ its FIB status; $n.t \leftarrow$ the table assigned to the corresponding route, $n.l, n.r \leftarrow$ its left and right children
  if n is a leaf node **then**
    $n.s = n.o$
  else if n is an internal node **then**
    Call $aggrInit(n.l)$ and $aggrInit(n.r)$
    if $n.l.s = n.r.s$ **then** ▷ Case 1: children nodes share the same next-hop
      $n.s \leftarrow n.l.s$
      $n.l.f \leftarrow n.r.f \leftarrow$*NON_FIB*
    else ▷ Case 2: children nodes have different next-hops
      $n.s = 0$
      if $n.l.s \neq 0$ **then** ▷ The left child of n is a point of aggregation
        $n.l.f \leftarrow$ *IN_FIB*, $n.l.t \leftarrow$ *DRAM*
        Push the prefix and the selected next-hop of $n.l$ into DRAM memory
      else
        $n.l.f =$ *NON_FIB*
      end if
      if $n.r.s \neq 0$ **then** ▷ The right child of n is a point of aggregation
        $n.r.f \leftarrow$ *IN_FIB*, $n.r.t \leftarrow$ *DRAM*
        Push the prefix and the selected next-hop of $n.r$ into DRAM memory
      else
        $n.r.f \leftarrow$ *NON_FIB*
      end if
    end if
  end if
end procedure

Algorithm 9 Post-order update in CFCA

procedure $postOrderUpdate(n, NH)$ ▷ $n \leftarrow$ node, $n.o \leftarrow$ its original next-hop; $n.l, n.r \leftarrow$ its left and right children; $n.rf$ its *REAL/FAKE* flag, NH is the new next-hop value.
  if $n.l.rf = FAKE$ **then**
    $n.l.o \leftarrow NH$
    $postOrderUpdate(n.l)$
  end if
  if $n.r.rf = FAKE$ **then**
    $n.r.o \leftarrow NH$
    $postOrderUpdate(n.r)$
  end if
  $setSelectedNextHop(n)$
  $setFIBstatus(n)$
end procedure

Algorithm 10 Bottom-up update in CFCA

procedure $bottomUp(n)$ ▷ $n \leftarrow$ node.
  repeat
    $n \leftarrow$ parent node of n
    $oldSelectedNextHop \leftarrow n.s$
    $setSelectedNextHop(n)$
    $setFIBstatus(n)$
  until $oldSelectedNextHop = n.s$
end procedure

Algorithm 11 Prefix fragmentation in CFCA

procedure FRAGMENTPREFIX(p,NH) ▷ p is the newly announced prefix; NH is p's next-hop.
    $n_0 \leftarrow$ The root node of the tree
    for each bit b of p **do**
        if n_0 is a leaf **then**
            $break$
        end if
        $n_0 \leftarrow b = 0 \ ? \ n_0.l : n_0.r$
    end for
    $l_0, l \leftarrow$Prefix lengths of $n_0.p$ and p
    for each level of the tree $\Delta \in (l_0, l)$ **do**
        Generate sibling *FAKE* nodes with the original next-hop of n_0
        Generate the *REAL* node n for p, $n.o \leftarrow NH$
        Generate the *FAKE* sibling node for n, $n.o \leftarrow n_0.o$
    end for
    return n_0
end procedure

Algorithm 12 Selecting next-hop in CFCA

procedure $setSelectedNextHop(n)$ ▷ $n \leftarrow$ node, $n.s \leftarrow$ its selected next-hop.
    if n is a leaf node **then**
        $n.s = n.o$
    else if n is an internal node **then**
        if $n.l.s = n.r.s$ **then**
            $n.s \leftarrow n.l.s$
        else
            $n.s = 0$
        end if
    end if
end procedure

Algorithm 13 Setting FIB status of a node in CFCA

```
1:  procedure setFIBstatus(n)                                   ▷ n ← node.
2:      if n is an internal node then
3:          if n.s ≠ 0 then                                     ▷ Case 1: n is the point of aggregation
4:              if n.l.f = IN_FIB then
5:                  Remove n.l from the data plane table n.l.t
6:                  n.r.f ← NON_FIB
7:              end if
8:              if n.r.f = IN_FIB then
9:                  Remove n.r from the data plane table n.r.t
10:                 n.r.f ← NON_FIB
11:             end if
12:         else if n.s = 0 then                                ▷ Case 2: n's descendants prefixes should be present in FIB
13:             if n.l.f = NON_FIB then
14:                 n.l.f ← IN_FIB, n.l.t ← DRAM
15:                 Push the prefix and the selected next-hop of n.l into the DRAM memory
16:             else
17:                 Push the selected next-hop update of the prefix of n.l into the data plane table n.l.t.
18:             end if
19:             if n.r.f = NON_FIB then
20:                 n.r.f ← IN_FIB, n.r.t ← DRAM
21:                 Push the prefix and the selected next-hop of n.r into the DRAM memory
22:             else
23:                 Push the selected next-hop update of the prefix of n.r into the data plane table n.r.t.
24:             end if
25:         end if
26:     end if
27: end procedure
```

Appendix E

Algorithms of VeriTable

For the description of each node field in the algorithms see Table E.1.

Name	Data Type	Description
$parent$	Node Pointer	Points to a node's parent node
l	Node Pointer	Points to a node's left child node if exists
r	Node Pointer	Points to a node's right child node if exists
$prefix$	String	Binary string
$length$	Integer	The length of the prefix, 0-32 for IPv4 or 0-128 for IPv6
$nexthop$	Integer Array	Next-hops of this prefix in $T_1...T_n$, size n
$type$	Integer	Indicates if a node is a $GLUE$ or $REAL$

Table E.1: Joint Patricia tree node's attributes.

Algorithm 14 Building a Joint PT T

procedure $BuildJointPT(T_1, T_2, ..., T_n)$
    Initialize a PT T with its head node
    Add prefix $0/0$ on its head node.
    Set default next hop values in the *Next Hops* array.
    for each table $T_i \in T_1, T_2, ..., T_n$ **do**
        for each entry e in T_i **do**
            Find a node n in T such as $n.prefix$ is a
             longest match for $e.prefix$ in T
            if $n.prefix = e.prefix$ **then**
                $n.nexthop_i \leftarrow e.nexthop$
                $n.type \leftarrow REAL$
            else
                Generate new node n'
                $n'.prefix \leftarrow e.prefix$
                $n'.nexthop_i \leftarrow e.nexthop$
                $n'.type \leftarrow REAL$
                Assume n has a child n_c
                if the overlapping portion of n_c and n' is longer than $n.length$ but shorter than $n'.length$ bits **then**
                    Generate a glue node g
                    $n'.parent \leftarrow g$
                    $n_c.parent \leftarrow g$
                    $g.parent \leftarrow n$
                    $g.type \leftarrow GLUE$
                    Set g as a child of n
                    Set n' and n_c as children of g
                else
                    $n'.parent \leftarrow n$
                    $n_c.parent \leftarrow n'$
                    Set n_c as a child of n'
                    Set n' as a child of n
                end if
            end if
        end for
    end for
end procedure

Algorithm 15 Forwarding Equivalence Verification. The initial value of $ancestor$ is $NULL$, and the initial value of $node$ is $T \rightarrow root$. For simplicity, we assume the root node is $REAL$.

1: **procedure** *VeriTable*($ancestor$, $node$)
2: **if** $node.type = REAL$ **then**
3: $ancestor$ = $node$ ▷ The closest ancestor node for a REAL node is the node istelf
4: **end if**
5: $l \leftarrow node.l$, $r \leftarrow node.r$
6: **if** $l \neq NULL$ **then**
7: **if** $l.type = REAL$ **then**
8: $InheritNextHops(ancestor, l)$ ▷ A REAL child node inherits next hops from the closest REAL ancestor to initialize "empty" next hops
9: **end if**
10: $LeftFlag \leftarrow VeriTable(ancestor, l)$ ▷ LeftFlag and RightFlag signify the existing leaks at the branches
11: **end if**
12: **if** $r \neq NULL$ **then**
13: **if** $r.type = REAL$ **then**
14: $InheritNextHops(ancestor, r)$
15: **end if**
16: $RightFlag \leftarrow VeriTable(ancestor, r)$
17: **end if**
18: **if** $l = NULL \wedge r = NULL$ **then**
19: $CompareNextHops(node)$ ▷ The leaf nodes' next hops are always compared; a verified node always returns the false LeakFlag.
20: $LeakFlag \leftarrow False$
21: **return** $LeakFlag$
22: **end if**
23: **if** $l \neq NULL \wedge l.length - node.length > 1$ **then**
24: $LeakFlag \leftarrow True$
25: **else if** $r \neq NULL \wedge r.length - node.length > 1$ **then**
26: $LeakFlag \leftarrow True$
27: **else if** $l = NULL \vee r = NULL$ **then**
28: $LeakFlag \leftarrow True$
29: **else if** $LeftFlag = True \vee RightFlag = True$ **then**
30: $LeakFlag \leftarrow True$
31: **end if**
32: **if** $LeakFlag = True \wedge node.type = REAL$ **then**
33: $CompareNextHops(node)$
34: $LeakFlag \leftarrow False$
35: **end if**
return $LeakFlag$
36: **end procedure**